TUTORIAL

The Open Motion Planning Library

By Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki

The open motion planning library (OMPL) is a new library for sampling-based motion planning, which contains implementations of many state-of-the-art planning algorithms. The library is designed in a way that it allows the user to easily solve a variety of complex motion planning problems with minimal input. OMPL facilitates the addition of new motion planning algorithms, and it can be conveniently interfaced with other software components. A simple graphical user interface (GUI) built on top of the library, a number of tutorials, demos, and programming assignments are designed to teach students about sampling-based motion planning. The library is also available for use through Robot Operating System (ROS).

Digital Object Identifier 10.1109/MRA.2012.2205651

Date of publication: 6 December 2012

Motion Planning

Robotic devices are steadily becoming a significant part of our daily lives. Search-and-rescue robots, service robots, surgical robots, and autonomous cars are examples of robots most of us are familiar with. Finding paths (motion plans) efficiently for such robots is critical for a number of real-world applications (Figure 1). For example, in urban search-and-rescue settings, a small robot may need to find paths through rubble and semicollapsed buildings to locate survivors. In domestic settings, it would be useful if a robot could, for example, put away kids' toys, fold the laundry, and load the dishwasher. Motion planning also plays an increasingly important role in robot-assisted surgery. For example, before a flexible needle is inserted or an incision is made, a path can be computed that minimizes the chances of harming vital organs. More generally, motion planning is the problem of finding a continuous path that connects a given start state of a robotic system to a given goal region for that system, such that the path satisfies a set of

1070-9932/12/$31.00©2012IEEE

constraints (e.g., collision avoidance, bounded forces, bounded acceleration). This article describes an open source software library for motion planning, designed for research, educational, and industrial applications.

Although most of the work done toward the development of algorithms that solve the motion planning problem comes from robotics and artificial intelligence [1]–[3], the problem can be viewed more abstractly as search in continuous spaces. As such, the applications of motion planning extend beyond robotics to fields such as computational biology [4]–[7] and computer-aided verification [8], among others.

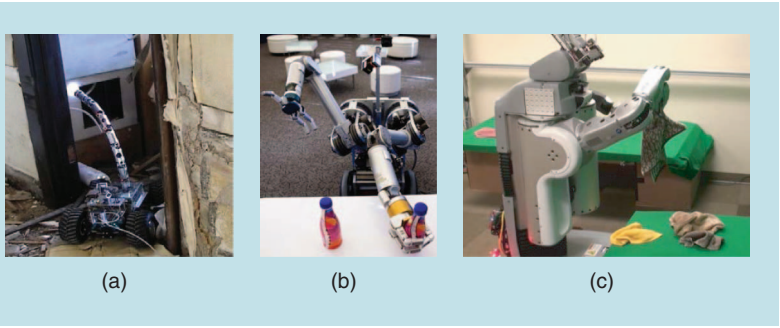

Figure 1. Real-world applications of motion planning. (a) An urban search-and-rescue robot from Carnegie Mellon University's (CMU's) Biorobotics Lab. (b) The HERB robot from CMU's Personal Robotics Lab picking up a bottle. (c) A PR2 robot folding laundry in the University of California at Berkeley's Robotics Learning Lab. Images used with permission from Prof. Choset, Prof. Srinivasa, and Prof. Abbeel, respectively.

Early results have shown the motion planning problem to be PSPACE-complete [9], and existing complete algorithms are difficult to implement and computationally intractable. For this reason, more recent efforts focus on approaches with weaker completeness guarantees. One of these approaches is that of sampling-based motion planning, which has been used successfully to solve difficult planning problems for robots of practical interest. Many sampling-based algorithms are probabilistically complete: a solution will eventually be found with probability one if one exists, but the nonexistence of a solution cannot be reported (see [10]–[13]).

Many of the core concepts in sampling-based motion planning are relatively easy to explain, but implementing sampling-based motion planning algorithms in a generic way is nontrivial. This article describes OMPL (http://ompl.kavrakilab.org), an open source C++ implementation of many sampling-based algorithms (including the Probabilistic Roadmap Method (PRM) [14], Rapidly-expanding Random Trees (RRT) [15], Kinodynamic Planning by Interior-Exterior Cell Exploration (KPIECE) [16], and many more) and the core low-level data structures that are commonly used. OMPL includes Python bindings that expose almost all functionality to Python users. This library is aimed at three different audiences:

- motion planning researchers
- robotics educators
- end users in the robotics industry.

In the following, we will characterize the needs of these different audiences.

Within the robotics community, it is often challenging to demonstrate that a new motion planning algorithm is an improvement over the existing methods, according to certain metrics. First, it is a substantial amount of work for a researcher to implement not only the new algorithm, but also one or more state-of-the-art motion planning algorithms to compare it against. Ideally, implementations of low-level data structures and subroutines used by these algorithms (e.g., proximity data structures) are shared, so that only differences of the high-level algorithm are measured. Second, for an accurate comparison, one needs a known set of benchmark problems. Finally, collecting various performance metrics for several planners with different parameter settings, running on several benchmark problems, and storing them in a way that facilitates easy analysis afterward is a nontrivial task. We, as developers of planning algorithms, have run into the above issues many times. We designed OMPL to help with all these issues, and make it easier to try out new ideas. Moreover, the library is designed in a way that facilitates contributions from other motion planning researchers and provides benchmarking capabilities to easily compare new planners against all other planners implemented in OMPL (see "Benchmarking with OMPL"). We have developed a streamlined process that gives contributing researchers appropriate credit and minimizes the burden of writing code that satisfies our library's application programmers interface (API). At the same time, our aim is to make such contributions easily available to users of OMPL. This is achieved by releasing the code under the Berkeley Software Distribution license (one of the least restrictive open source licenses), releasing frequent updates, and making the code available through a public repository. To foster a community of OMPL users and developers, we have set up a mailing list, a blog, and a Facebook page.

For robotics educators, we have designed a series of exercises or projects around OMPL aimed at undergraduate students. These exercises help students to realize the complexity of motion planning in practice, to develop an understanding of how sampling-based motion planning algorithms work, and to learn evaluation of the performance of planners. We have also designed open-ended projects for undergraduate and graduate students. OMPL is structured to have a clear mapping between the motion planning concepts used in the literature and the classes that are defined in the implementation. The separation between abstract base classes that only specify the interface and derived classes that implement the specified functionality also helps students to understand general concepts in motion planning before focusing on details.

From the beginning, OMPL was intended to be useful in practical applications. This requires that planning algorithms

can solve motion planning problems for systems with many degrees of freedom at interactive speeds. An additional requirement is the ability to cleanly integrate OMPL with other software components on a robot, such as perception, kinematics, and control. Through a collaboration with Willow Garage, Menlo Park, California, OMPL is integrated within ROS [17] and serves as the motion planning back-end for the arm planning software stack. The availability of OMPL in ROS makes it easy for end users in the robotics industry to stay up-to-date with advances in sampling-based motion planning.

Background

There has been much work done on both algorithm development and software development for motion planning. This article only discusses aspects pertaining to sampling-based motion planning.

Sampling-Based Motion Planning Definitions

Sampling-based motion planning algorithms relaxed completeness guarantees and demonstrated that many interesting problems can be solved efficiently in practice, despite the theoretically high complexity of the problems [2], [3]. The fundamental idea of sampling-based motion planning is to approximate the connectivity of the search space with a graph structure. The search space is sampled in various ways, and selected samples end up as the vertices of the approximating graph. Edges in the approximating graph denote valid path segments.

There are two key considerations in the construction of the graph approximation: the probability distribution used for sampling states and the strategy for generating edges. An enormous amount of research has been performed toward the development of efficient algorithms that account for these issues [18].

We will not go into the details of various sampling-based motion planning algorithms, as such details can be found elsewhere [2], [3]. Instead, we describe the common components sampling-based algorithms typically depend on, as these relate to the implementations of such algorithms:

- *State Space*: Points in the state space (or configuration space) fully describe the state of the system being planned for. For a free-flying rigid body, the state space consists of all translations and rotations, while for a manipulator with n rotational joints, the state space can be modeled by an n-dimensional torus.
- *Control Space*: A control space represents a parameterization of the space of controls. This is only required for systems with dynamics. For most systems of practical interest, one can think of the control space for a system with m controls simply as a subset of $\mathbb{R}^m$. For geometric planning, no controls are used.
- *Sampler*: A sampler is needed to generate different states from the state space. For control-based systems, a separate sampler is needed for sampling different controls. Some planning algorithms (e.g., [12], [16]) only require a control sampler and do not need a state sampler.
- *State Validity Checker*: A state validity checker is a routine that distinguishes the valid part of the state space from the invalid part of the state space. For example, a state validity checker can check for collisions and whether velocities and accelerations are within certain bounds.

Benchmarking with OMPL

A seemingly simple but often ignored part of motion planning software is benchmarking planning code. OMPL includes benchmarking capabilities (through a class called Benchmark) that can be simply dropped in and applied to existing planning contexts. In very simple terms, a Benchmark object runs a number of planners multiple times on a user-specified planning context. Although simple, this code automatically keeps track of all the used settings and takes all the possible measurements during planning (currently, tens of parameters are recorded for every single motion plan). The recorded information is logged and can be postprocessed using a Python script included with OMPL. The script can produce MySQL databases with all experiment data so that the user can write their own queries later on, but it can also automatically generate plots for all of the performance metrics. For real- and integer-valued measurements, it generates box plots: plots that include information about the median, confidence intervals and outliers. An example is shown in Figure S1. For binary-valued measurements, it generates bar plots. A more elaborate example of what can be done with the Benchmark class can be found at http://plannerarena.org, a Web site currently being developed to establish standard benchmark problems and report performance metrics for various planners on those problems.

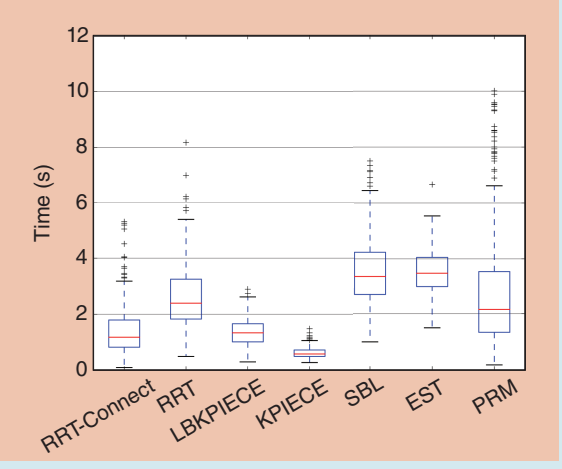

Figure S1. A sample box plot generated by OMPL's benchmark script.

- *Local Planner*: When planning with controls, the local planner is a mean of computing the evolution of the robotic system forward (and sometimes backward) in time. When planning solely under geometric constraints, the local planner often performs interpolation between states in the state space.

Software Packages for Motion Planning

Several other packages for motion planning are available. Some, such as the Motion Strategies Library (MSL, http://msl.cs.uiuc.edu), the Motion Planning Kit (MPK, http://robotics.stanford.edu/~mitul/mpk), and VIZMO++ [19] are no longer maintained. KineoWorks (http://www.kineocam.com) provides commercial motion planning software for academic research and industrial applications. In 2007, our group released the Object-Oriented Programming System for Motion Planning (OOPSMP) [20], which is also no longer maintained.

Another software package that is complementary to OMPL is OpenRAVE [21]. OpenRAVE is open source, actively developed, and it is widely used. It is important to understand the difference in design philosophy behind OMPL and OpenRAVE. OpenRAVE is designed to be a complete package for robotics. It includes, among other things, geometry representation, collision checking, grasp planning, forward and inverse kinematics for several robots, controllers, motion planning algorithms, simulated sensors, and visualization tools. OMPL, on the other hand, was designed to focus completely on sampling-based motion planning with a clear mapping between theoretical concepts in the literature and abstract classes in the implementation. This high level of abstraction makes it easy to integrate OMPL with a variety of front-ends and other libraries. Some integration examples are described in section describing the integration with other robotics software. To some extent, the integration with ROS [17] gives a user many of OpenRAVE's features that are purposefully not included in OMPL. It may also be possible to use OMPL as a motion planning plug-in in OpenRAVE. As a result of the narrower focus in OMPL, we have been spending more resources on implementing a much broader variety of sampling-based algorithms than what is currently available in OpenRAVE, as well as benchmarking capabilities to facilitate a thorough comparison of existing and future sampling-based motion planners.

Relationship with Other Robotics Software

There have also been many efforts to create robot simulators such as Player/Stage [22], Player/Gazebo [23], Webots [24], and MORSE [25]. Microsoft Robotics Developer Studio [26] also contains a robot simulator. Typically, such simulators do not include motion planning algorithms, but they can provide a controlled simulated environment to test motion planners in various environments, on various robots with different sensing and communication capabilities. They often simulate the dynamics of the world (including the robots themselves) using physics engines such as Bullet (http://bulletphysics.org) and the open dynamics engine (ODE, http://ode.org), among others.

Hardware platforms typically require complex software configurations and use various forms of middleware to accommodate this requirement (e.g., ROS [17], Orocos (http: //www.orocos.org), OpenRTM-aist [27], OPRoS [28], Yarp [29]). Such software systems typically include their own visualization system, collision checking, etc. OMPL fits naturally and easily into such systems as it only provides sampling-based motion planning and its abstract interface should accommodate a variety of low-level implementations.

Many of the core concepts in sampling-based motion planning are relatively easy to explain, but implementing sampling-based motion planning algorithms in a generic way is nontrivial.

Conceptual Overview of OMPL

OMPL is intended for use in research and education, as well as in industry. For this reason, the main design criteria for OMPL were as follows:

1) *Clarity of Concepts*: OMPL was designed to consist of a set of components as indicated in Figure 2, such that each component corresponds to known concepts in sampling-based motion planning.
2) *Efficiency*: OMPL has been implemented entirely in C++ and is thread-safe.
3) *Simple Integration with Other Software Packages*: To facilitate the integration with other software libraries, OMPL offers abstract interfaces that can be implemented by the "host" software package. Furthermore, the dependencies of OMPL are minimal: only the Boost C++ libraries are required. Optionally, OMPL can be compiled with Python bindings, which facilitates integration with Python modules.
4) *Straightforward Integration of External Contributions*: We strive for minimalist API constraints for planning algorithms, so that new contributions can be easily integrated.

As opposed to all other existing motion planning software libraries, OMPL does not include a representation of workspaces or of robots; as a result, it also does not include a collision checker or any means of visualization. OMPL is reduced to only motion planning algorithms. The advantage of this minimalist approach is that it allows us to design a library that can be used for generic search in high-dimensional continuous spaces subject to complex constraints. Instead of defining valid states as collision-free, which would require a specific geometric representation of the environment and robot as well as support for a specific collision

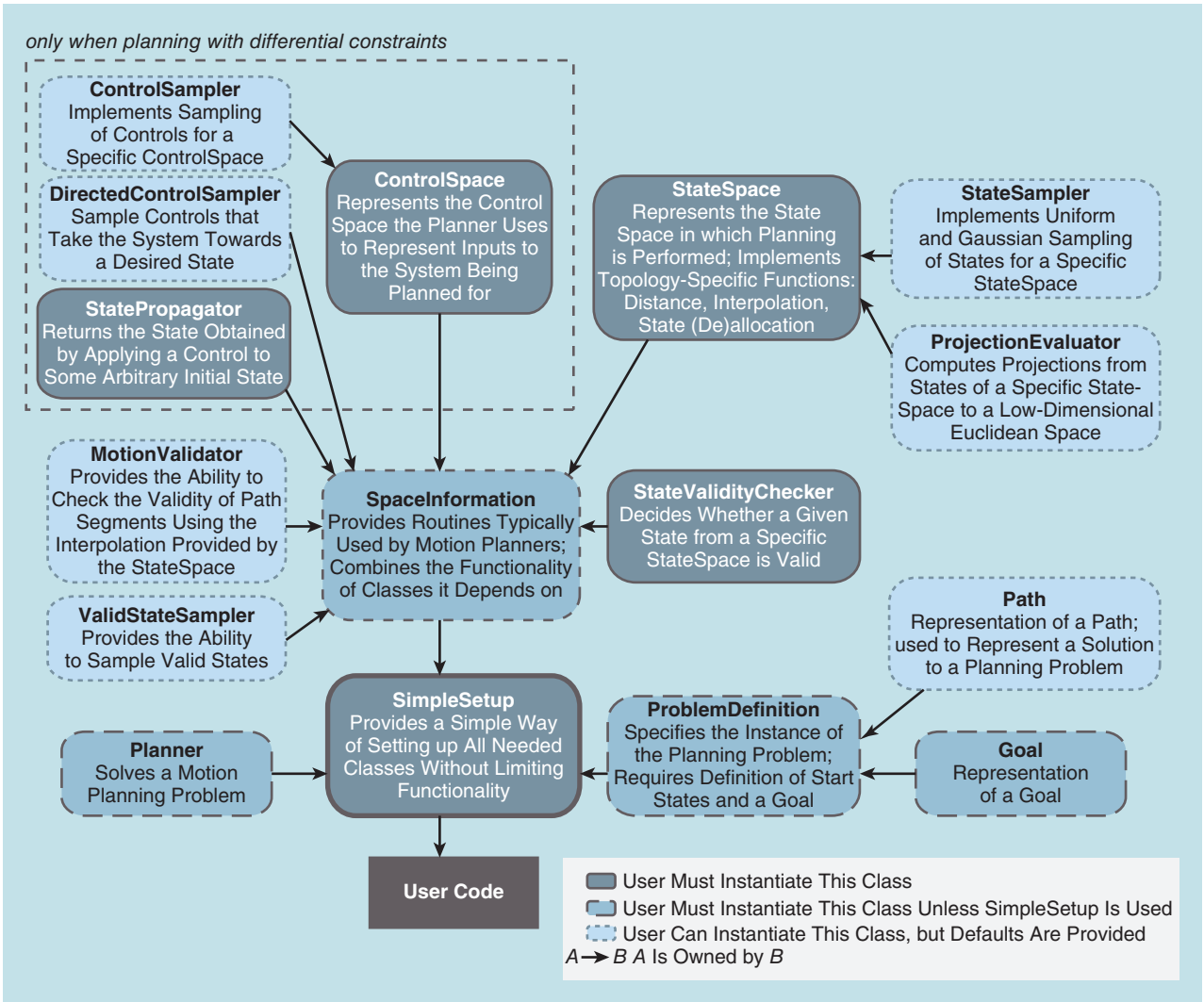

Figure 2. Overview of OMPL structure. Class names correspond to well-understood concepts in sampling-based motion planning. More detailed documentation is available at http://ompl.kavrakilab.org.

checker, OMPL leaves the definition of state validity completely up to the user (or the software package in which OMPL is integrated; see the section about the relationship with other robotics software). This gives the user an enormous design freedom: the user can defer collision checking to a physics engine, write a state sampler that constructs only valid states, or define state validity in completely arbitrary ways that may or may not depend on geometry.

To make OMPL as easy to use as possible, various parameters needed for tuning sampling-based motion planners are automatically computed. The user has the option to override defaults, but that is not a requirement.

Implementation of Core Concepts

In the following we will give an overview of the implementation of the core motion planning concepts in OMPL. Figure 2 gives a high-level overview of the main classes and their relationships. We will use the following notation. Classes are written in a sans-serif font (e.g., StateSpace), while methods and functions are written in a monospaced font [e.g., `isSatisfied()`]. For conciseness, the arguments to methods and functions are omitted.

States, Controls, and Spaces

To maximize the range of application for the included planning algorithms, OMPL represents the search spaces, that is, the state spaces (StateSpace), in a generic way. State spaces include operations on states such as distance evaluation, test for equality, interpolation, as well as memory management for states: (de)allocation and copying. Additionally, each state space has its own storage format for states, which is not exposed outside the implementation of the state space itself. To operate on states, the planning algorithms implemented in OMPL rely only on the generic functionality offered by state spaces. This approach enables planning algorithms in OMPL to be applicable to any state spaces that may be defined, as long as the expected generic functionality is provided.

Furthermore, OMPL includes a means of combining state spaces using the class CompoundStateSpace. A combined state space implements the functionality of a regular state space on top of the corresponding functionality from the maintained set of state spaces. This allows trivial construction of more complex state spaces from simpler ones. For example SE3StateSpace [the space of rigid body transformations in three-dimensional (3-D)] is just a combination of SO3StateSpace (the space of rotations) and RealVectorStateSpace (the space of translations). Instances of CompoundStateSpace can be constructed at run time, which is necessary for constructing a state space from an input file specification, as is done, for example, in ROS. For a mobile manipulator, one could construct a CompoundStateSpace with the two arms and the mobile base as substate spaces. An arm typically has a number of rotational joints and can be modeled by either a RealVectorStateSpace (if the joints have limits) or a CompoundStateSpace with copies of SO(2). The state space for the base can simply be SE(2) (the space rigid body transformations in the plane).

State spaces optionally include specifications of projections to Euclidean spaces (ProjectionEvaluator). Low-dimensional Euclidean projections are used by several sampling-based planning algorithms (e.g., KPIECE [16], SBL [30], EST [12]) to guide their search for a feasible path, as it is much easier to keep track of coverage (i.e., which areas have been sufficiently explored and which areas should be explored further) in such low-dimensional spaces.

In addition to states and state spaces, some algorithms in OMPL require a means to represent controls. Control spaces (ControlSpace) mirror the structure of state spaces and provide functionality specific to controls, so that planning algorithms can be implemented in a generic way. The only available implementations of control spaces are the Euclidean space and a space for discrete modes, because so far there has not been a need for control spaces with more complex topologies. However, the API allows one to define such control spaces.

State Validation and Propagation

Whether a state is valid or not depends on the planning context. In many cases, state validity simply means that a robot is not in collision with any obstacles, but in general any condition on a state can be used. In OMPL.app (see the section on OMPL.app: A GUI for OMPL) we have predefined a state validity checker for rigid body motion planning. We have also implemented a state validity checker that uses the ODE (see the section on motion planning using a physics engine). If these built-in state validity checkers cannot be used for the system of interest, a user needs to implement their own. Based on a given state validity checker, a default MotionValidator is constructed that checks whether the interpolation between two states at a certain resolution produces states that are all valid. However, it possible to plug in a different MotionValidator. For example, one might want to add support for continuous collision checking, which can adaptively check for collisions and provide exact guarantees for state validity [31].

For planning with controls, a user needs to specify how the system evolves when certain controls are applied for some period of time starting from a given state. This is called state propagation in OMPL. In the simplest case, a state propagator is essentially a lightweight wrapper around a numerical integrator for systems of the form $\dot{q} = f(q, u)$, where q is a state vector and u a vector of controls. To facilitate planning for such systems, we have implemented generic support for ODE solvers and we have integrated Boost.Odeint [32], a new library for solving ODEs. Given a user-provided function that implements $f(q, u)$ for the system of interest, OMPL can plan for such systems. Alternatively, one can use variational integrators [33], or a physics engine to perform state propagation.

Motion Planning Using a Physics Engine

OMPL has built-in support for using the ODE physics engine. Support for other physics engines, such as Bullet, is planned for a future release. We expect that the approach described below can be followed for these physics engines (and others) as well.

The ODE state space consists of the state spaces of the robot and any movable objects in the environment. The user specifies which joints are controlled by the planner and maps those to a ControlSpace. The user can also specify which collisions are allowed (e.g., contact with the support plane) and which ones are not (such as driving into a wall). This simple setup allows one to plan for systems that are difficult to describe with differential equations. The user does not need to worry about all the different possible contact modes that occur when a car drives off a ramp (Figure S2) or when a robot pushes one or more obstacles (Figure S3).

Figure S2. A car-like robot driving off a ramp.

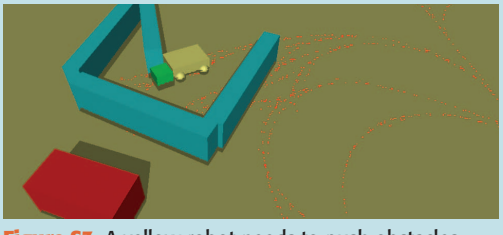

Figure S3. A yellow robot needs to push obstacles to get to its goal.

Samplers

The fundamental operation that sampling-based planners perform is sampling the space that is explored. Additionally, when considering controls in the planning process, sampling controls may be performed as well.

To support sampling functionality, OMPL includes four types of samplers: state space samplers (StateSampler), valid state samplers (ValidStateSampler), control samplers (ControlSampler), and directed control samplers (DirectedControlSampler).

State space samplers are implemented as part of the StateSpace they can sample, since they need to be aware of the structure of the states in that space. For instance, uniformly sampling 3-D orientations is dependent on their parameterization. Three sampling distributions are implemented by every state space sampler: uniform, Gaussian and uniform in the vicinity of a specified point. This first sampler is necessary to sample over the entire space, but the latter two are used for sampling states near a previously generated state. This is the most basic level of sampling.

Within the robotics community, it is often challenging to demonstrate that a new motion planning algorithm is an improvement over the existing methods, according to certain metrics.

Previous work has shown that the strategy used for sampling valid states in the state space significantly influences runtime of many planning algorithms [34]. Valid state samplers provide the interface for implementing different sampling strategies. The probability distribution of these samplers depends on the algorithm used and is not imposed as part of the API. The implementation of valid state samplers relies on the existence of a state space sampler and a state validator (StateValidityChecker). A common approach to constructing valid state samplers is to repeatedly call a state space sampler until the state validator returns true. Several valid state samplers have been implemented in OMPL: for example, a uniform valid state sampler (UniformValidStateSampler), two samplers (GaussianValidStateSampler, ObstacleBasedValidStateSampler) that generate valid samples near invalid ones (which is often helpful in finding paths through narrow passages [35], [36]).

When considering controls in the planning process, a means to generate controls is also necessary. This functionality is attained using control samplers, which are implemented as part of the control spaces (ControlSpace) they represent. Additionally, a notion of direction is also important in some planners: controls that take the system towards a particular state are desired, rather than simply random controls. This functionality is achieved through the use of directed control samplers (derived from the DirectedControlSampler class).

Goal Representations

OMPL uses a hierarchical representation of goals. In the most general case, a Goal can be defined by the `isSatisfied()` function that when given a state, reports whether that state is a goal state or not. While this very general implicit representation is possible, it offers planners indication of how to reach the goal region. For this reason, `isSatisfied()` optionally reports a heuristic distance the goal region, which is not required to be a metric.

GoalRegion is a refinement of the general Goal representation, which explicitly specifies the distance to the goal using a `distanceGoal()` function. The `isSatisfied()` function is then defined to return true when `distanceGoal()` reports distances smaller than a user set threshold. GoalRegion is still a very general representation but allows planners to bias their search towards the goal. A refinement of GoalRegion is GoalSampleableRegion, one which additionally allows drawing samples from the goal region. GoalState and GoalStates are concrete implementations of GoalSampleableRegion.

For practical applications it is often possible to sample the goal region, but the sampling process may be relatively slow (e.g., when using numerical inverse kinematics solvers). For this reason a refinement of GoalStates is defined as well: GoalLazySamples. This refinement continuously draws samples in a separate sampling thread, and allows planners to draw samples from the goal region without waiting, after at least one sample has been produced by the sampling thread.

Planning Algorithms

OMPL includes two types of motion planners: ones that do not consider controls when planning and ones that do. If a planning algorithm can be used to plan both types of motions, with and without controls (e.g., RRT [15]), two separate implementations are provided for that algorithm: one for each type of computed motion. This choice was made for efficiency reasons. With additional levels of abstraction in the implementation, it would have been possible to avoid separate implementations, [20]. The downside would have been that the implementation of planners would have had to follow a strict structure, which makes the implementation of new algorithms more difficult and possibly less efficient.

For purely geometric planning (i.e., controls are not considered), the solution path is constructed from a finite set of segments, and each segment is computed by interpolation between a pair of sampled states (PathGeometric). Several geometric planning algorithms are implemented in OMPL, including KPIECE [16], bidirectional KPIECE, bidirectional lazy KPIECE, RRT [15], RRT-connect [37], lazy RRT, SBL [30], EST [12], and PRM [14]. The lazy variants listed above defer state validity checking in the

manner described in [38]. In addition, there are multithreaded versions of RRT and SBL.

When controls are considered, the solution path is constructed from a sequence of controls (PathControl). Controlbased planners are typically used when motion plans need to respect differential constraints as well. Several algorithms for planning with differential constraints are implemented in OMPL as well, including KPIECE, SyCLoP [39], EST, and RRT.

An Example

Figure 3 shows the complete code necessary for planning the motion of a rigid body between two states in Python. The corresponding C++ code would look almost identical. The steps taken in the code are: instantiate the space to plan in [SE(3), line 6], create a simple planning context (using SimpleSetup, line 13), specify a function that distinguishes valid states (lines 15 and 16), specify the input start and goal states (lines 18–26), and finally, compute the solution (line 27). The SimpleSetup class initializes instantiations of the core motion planning classes shown in Figure 2 with reasonable defaults, which can be overridden by the user if desired.

Essentially, the execution of the code can be reduced to three simple steps: 1) specify the space in which planning is to be performed, 2) specify what constitutes a valid state, and 3) specify the input start and goal states. Such simple specifications are desirable for many users who simply want motion planning to work without having to select problem-specific parameters, or different sampling strategies, or different planners, etc. This capability is made possible by OMPL's automatic computation of planning parameters. In the example above, a planner is automatically selected based on the specification of the goal and the space to plan in. The selected planner is then automatically configured by computing reasonable default settings that depend on the planning context. If a user decides to choose their own planner, or set their own parameters, OMPL allows the user to do so completely—no parameters are hidden.

Integration with Other Robotics Software

It is straightforward to integrate OMPL with other robotics software. In the following we will present two case studies that highlight different use cases.

OMPL.app: A GUI for OMPL

We have created a graphical front end for OMPL called OMPL.app. This front end was created for three reasons:

1) to provide novice users (such as students in a robotics class) with an easy-to-use interface to play with several motion planning algorithms and apply them to several example rigid body motion planning problems
2) to demonstrate the integration of OMPL with third-party libraries for collision checking and loading of 3-D meshes, and a GUI toolkit

```
1   def is StateValid (state):
2         # Some arbitrary condition on the state
3         # (typically collision checking)
4         return state.getX () < .6
5
6   space = SE3StateSpace ()
7   # set the state space bounds
8   bounds = ob.RealVectorBounds (3)
9   bounds.setLow(−1)
10  bounds.setHigh (1)
11  space.setBounds (bounds)
12
13  ss = SimpleSetup (space)
14  # specify user-defined callback function
15  ss.setStateValidityChecker (
16        ob.StateValidityCheckerFn (isStateValid))
17
18  start = State (space)
19  goal = State (space)
20  # we can pick random start states...
21  start.random ()
22  goal.random
23  # ... or set specific values
24  start ().setX (.5)
25
26  ss.set StartAndGoalStates (start, goal)
27  solved = ss.solve (1.0)
28  if solved:
29        print ss.getSolutionPath()
```

Figure 3. Solving a motion planning problem with OMPL in Python. A C++ implementation would look almost identical.

3) to allow for easy benchmarking of new and existing planners on rigid body motion planning problems using a command line tool (see "Benchmarking with OMPL"). We will go on to elaborate on these reasons.

The graphical interface of OMPL.app is shown in Figure 4. A user can load meshes that represent the environment and a robot, define start and goal states, and click on the "Solve" button to obtain a solution. If a solution is found, it is played back by animating the robot along the found path. By unchecking the "Animate" button, several states along the path are shown simultaneously. It is also possible to show the states that were explored by the

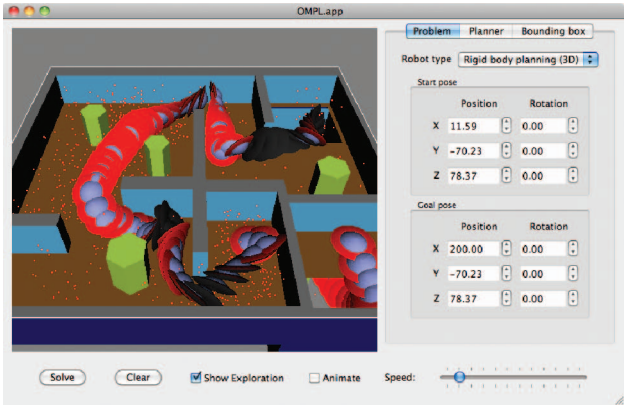

Figure 4. The OMPL.app graphical interface. A solution path for a free-flying UFO robot is shown. Red dots are positions of sampled states.

planner, which can be helpful in tuning planner parameters or selecting the appropriate planning algorithm for a particular problem. By default, the program assumes that a user wants to plan for a free-flying 3-D rigid body [i.e., the state space is SE(3)], but one can also plan in SE(2). We have also predefined a number of common robot types that require controls: a blimp, a quadrotor, a simple kinematic car, a Reeds-Shepp car, a Dubins car, and a second-order car. For each robot type the appropriate planners can be selected in the Planner tab (otherwise, a default one will be automatically selected). Once a planner is chosen, its parameters can be tuned if desired. Finally, the user can adjust the bounding box for the robot's position. By default this is the bounding box of the environment mesh. We have included a number of common benchmark problems, which allow users to develop a basic understanding of which types of problems are hard to solve.

Applications of motion planning extend beyond robotics to fields such as computational biology and computer-aided verification.

The OMPL.app program is also an illustrative example for software developers who want to integrate with third party libraries or their own code. OMPL.app consists of three parts: 1) a C++ library that contains the bindings to third-party libraries, 2) a set of command line demos that highlight significant features of this library, and 3) the GUI itself. The library adds functionality to load meshes in a wide variety of formats using the Open Asset Importer Library (Assimp, http://assimp.sf.net). Users can thus create models of environments and robots in programs such as SolidWorks, 3DS Max, Blender, and SketchUp, and use them in OMPL.app. A large collection of models is also available through the Google 3-D Warehouse. The OMPL.app library also adds collision checking support using the PQP library [40] and FCL library [41]. The internal representation of geometry is decoupled from the graphical rendering, so that the collision checking can also be used in nongraphical applications. The user interface is written completely in Python. The code consists almost completely of creating the user interface elements, connecting them with the appropriate library function calls, and displaying the results.

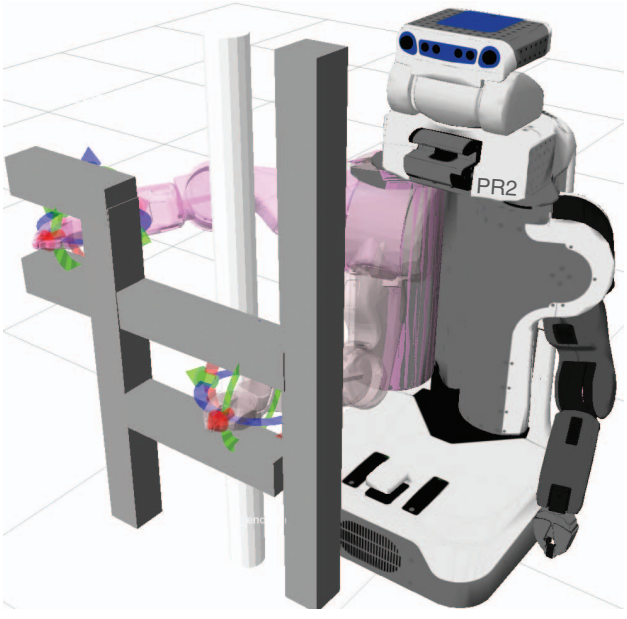

Figure 5. With the ROS rviz visualizer robot poses can be easily configured using OMPL to find feasible paths between poses.

The GUI is also a useful tool to prepare motion planning problems for benchmarking. The GUI can save the complete specification of a problem to a simple text file. The user can then add a list of planner names to this file, along with planner parameter settings, the number of runs per planner, and a time limit for each run, among other data. This configuration file can be given as input to a simple command line program that can perform the actual benchmarking. Usually, the total time required to get statistically significant benchmark results is too long for interactive use for all but the simplest problems, which is why the benchmarking is not directly accessible from the GUI.

It should be relatively straightforward for an experienced programmer to use a different input file parser, a different collision checking library, or different GUI toolkit by mimicking the structure of the OMPL.app library.

Integration with ROS

We expect that many end users in industry and robotics research will use OMPL through its ROS interface. This interface was created by Sachin Chitta and Ioan Şucan, and provides ROS-specific implementations for the abstract base classes OMPL defines. We describe the steps an end-user would need to take to plan motions for a given robot that runs ROS. The PR2 from Willow Garage will be used in the scenario described in the following, but the steps are in fact not specific to the PR2, and apply to any robot that can run ROS.

If a user wants to plan motions for a PR2, they first need to create a model of the geometry and kinematics of the PR2. Within ROS, there is a standard for storing such a model called the unified robot description format (URDF, http://ros.org/wiki/urdf). This XML-based format combines kinematic information with references to files containing meshes of the different robot components. For the PR2 and many other robots, the URDF files already exist (see http://www.ros.org/wiki/Robots). It is often neither desirable nor necessary to plan for all degrees of freedom listed in a URDF file simultaneously. The second step therefore consists of defining one or more groups of joints. Information about the joints to plan for is taken from the URDF and a `StateSpace` representation for OMPL is constructed. The meshes indicated by the URDF document are used to construct a `StateValidityChecker` class.

On top of these classes, a SimpleSetup class can also be defined, thus making it possible to solve planning problems. The user can also define parameters specific to different planning algorithms, but there is no requirement to do so. A configuration wizard included in ROS can make the setup easier. The third step is to define motion planning problems for the PR2. This can be done in a variety of ways: directly calling planning functions, using ROS-specific APIs, or through visualization tools such as those shown in Figure 5.

Thus, we have described a very basic workflow of planning paths using OMPL in ROS. The ROS-OMPL interface has many more advanced features. First, motion planning problems do not necessarily need to be specified by the user, but can be specified programmatically (e.g., as part of a sense-plan-act loop in conjunction with other components in ROS). Second, different types of state space parameterizations are possible: 1) joint-space representations of the robot, where the robot's degrees of freedom are compounded into different state spaces: $\mathbf{R}^n$ for sequences of single degree of freedom joints with joint limits, *SO(2)* for continuous joints, *SE(2)* for robots moving in plane and *SE(3)* for robots moving in space, and 2) work-space representations of the robot, where for example, the pose of an arm's end-effector is represented as an *SE(3)* state, and the interpolation capability of the *SE(3)* state space is overridden to use inverse kinematics. Third, the ROS interface allows the user to specify complex constraints such as keeping transported objects upright or keeping them within view. Generating states that are in the desired goal region is done in parallel with the execution of the rest of the planning algorithm. The interface also automatically incorporates the geometry of attached objects during planning by attaching carried objects to the kinematic model of the robot.

The ROS interface to OMPL allows users to interact with motion planners in a simple manner. Only the set of joints the user wants to plan for (usually grouped and referred to by the name of the group) and a specification of the goal are needed. The goal can be specified, for example, as a bounding box in the joint space, or a desired link pose. We believe that this functionality will allow for the widespread use of OMPL in a broad variety of settings.

Discussion

We have described OMPL, an open source general-purpose library for sampling-based motion planning. Thanks to its integration with ROS, it can be used on a wide variety of hardware platforms, and currently serves as the motion planning back end for the ROS manipulation software stack (also known as MoveIt! in future releases of ROS [42]). However, OMPL does not depend on ROS, and can be used independently. OMPL.app includes a graphical front end for OMPL and serves as an example of how OMPL can be integrated with third-party libraries.

One of the target applications of OMPL is in robotics education. The graphical front end provides a gentle introduction to the complexity of motion planning: without writing any code, students can solve motion planning problems using different planners, vary the parameters used for planning, and perform extensive benchmarking experiments. Through many demo programs and tutorials, students should get quickly up to speed and develop new planning algorithms or alternate implementations of abstract APIs.

The ROS interface to OMPL allows users to interact with motion planners in a simple manner.

We encourage contributions from other researchers. In fact, we are already working with other research groups on incorporating their algorithms into OMPL. Within our own group, OMPL has proven to be useful for performing conformational search for macromolecular ensembles. Here, its generality has paid off significantly. We can use Rosetta—a standard molecular modeling package—to create a new state space for molecules, and use Rosetta's sampling capabilities while performing a search for biophysically plausible configurations of molecules using OMPL.

Long-term success depends on adoption and support by the robotics community. Through continued development in our group and contributions from others, we expect OMPL to become a very useful tool for motion planning researchers, users in the robotics industry, and students who want to learn more about sampling-based motion planning.

Acknowledgments

This work was supported in part by Willow Garage and NSF CCLI under Grant DUE 0920721 and Grant IIS 0713623. Willow Garage was instrumental in initiating and supporting this effort. The development of OMPL has also benefited from previous efforts by the Kavraki Laboratory at Rice University to develop a motion planning software package, in particular OOPSMP. We are indebted to all other Kavraki Laboratory members, past and present, for providing code, inspiration, and feedback. Previous work by Erion Plaku, Kostas Bekris, and Andrew Ladd in particular has been influential in the design of OMPL. The authors would also like to thank Sachin Chitta and Gil Jones from Willow Garage for the development of the ROS bindings for OMPL and helpful discussions.

References

[1] J.-C. Latombe, *Robot Motion Planning*. Boston, MA: Kluwer 1991.

[2] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations*. New York: MIT Press, 2005.

[3] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, (2006). [Online]. Available: http://msl.cs.uiuc.edu/planning/

[4] S. Thomas, X. Tang, L. Tapia, and N. M. Amato, "Simulating protein motions with rigidity analysis," *J. Comput. Biol.*, vol. 14, no. 6, pp. 839–855, 2007.

[5] B. Raveh, A. Enosh, O. Schueler-Furman, and D. Halperin, "Rapid sampling of molecular motions with prior information constraints," *PLoS Comput. Biol.*, vol. 5, no. 2, p. e1000295, Feb. 2009.

[6] J. Cortés, S. Barbe, M. Erard, and T. Siméon, "Encoding molecular motions in voxel maps," *IEEE/ACM Trans. Comp. Biol. Bioinf.*, vol. 8, no. 2, pp. 557–563, Apr. 2010.

[7] N. Haspel, M. Moll, M. L. Baker, W. Chiu, and L. E. Kavraki, "Tracing conformational changes in proteins," *BMC Structural Biol.*, vol. 10, no. 1, p. S1, 2010.

[8] E. Plaku, L. E. Kavraki, and M. Y. Vardi, "Hybrid systems: From verification to falsification by combining motion planning and discrete search," *Formal Methods Syst., Design*, vol. 34, pp. 157–182, 2009.

[9] J. Canny, *The Complexity of Robot Motion Planning.* Cambridge, MA: MIT Press, 1988.

[10] L. E. Kavraki, J.-C. Latombe, R. Motwani, and P. Raghavan, "Randomized query processing in robot path planning," in *Proc. 27th Annual ACM Symp. Theory Computing*, 1995, pp. 353–362.

[11] J. Barraquand, L. E. Kavraki, J.-C. Latombe, R. Motwani, T.-Y. Li, and P. Raghavan, "A random sampling scheme for path planning," *Int. J. Robot. Res.*, vol. 16, no. 6, pp. 759–774, 1997.

[12] D. Hsu, J.-C. Latombe, and R. Motwani, "Path planning in expansive configuration spaces," *Int. J. Comput. Geometry Appl.*, vol. 9, nos. 4–5, pp. 495–512, 1999.

[13] A. M. Ladd and L. E. Kavraki, "Measure theoretic analysis of probabilistic path planning," *IEEE Trans. Robot. Autom.,* vol. 20, no. 2, pp. 229–242, Apr. 2004.

[14] L. E. Kavraki, P. Švestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Trans. Robotics Autom.*, vol. 12, no. 4, pp. 566–580, Aug. 1996.

[15] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," *Int. J. Robot. Res.*, vol. 20, no. 5, pp. 378–400, May 2001.

[16] I. A. Şucan and L. E. Kavraki, "A sampling-based tree planner for systems with complex dynamics," *IEEE Trans. Robot.,* vol. 28, no. 1, pp. 116–131, 2012.

[17] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: An open-source robot operating system," in *Proc. ICRA Workshop Open Source Software*, 2009, pp. 1–6.

[18] I. A. Şucan and L. E. Kavraki, "On the implementation of single-query sampling-based motion planners," in *Proc. IEEE Int. Conf. Robotics Automation*, May 2010, pp. 2005–2011.

[19] A. V. Estrada, J. M. Lien, and N. M. Amato, "VIZMO++: A visualization, authoring, and educational tool for motion planning," in *Proc. 2006 IEEE Int. Conf. Robotics Automation*, pp. 727–732.

[20] E. Plaku, K. E. Bekris, and L. E. Kavraki, "OOPS for motion planning: An online open-source programming system," in *Proc. 2007 IEEE Int. Conf. Robotics Automation*, Rome, Italy, pp. 3711–3716.

[21] R. Diankov, "Automated construction of robotic manipulation programs," Ph.D. dissertation, Carnegie Mellon University, Robotics Institute, Aug. 2010. [Online]. Available: http://www.programmingvision.com/rosen diankov thesis.pdf

[22] B. Gerkey, R. Vaughan, and A. Howard, "The Player/Stage project: Tools for multi-robot and distributed sensor systems," in *Proc. 11th Int. Conf. Advanced Robot.*, 2003, pp. 317–323.

[23] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *Proc. 2004 IEEE/RSJ Int. Conf. Intelligent Robots Systems,* pp. 2149–2154.

[24] O. Michel, "Webots™: Professional mobile robot simulation," *Int. J. Adv. Robot. Syst.*, vol. 1, no. 1, pp. 39–42, 2004. [Online]. Available: http://www.cyberbotics.com/

[25] G. Echeverria, N. Lassabe, A. Degroote, and S. Lemaignan, "Modular open robots simulation engine: MORSE," in *Proc. IEEE Int. Conf. Robotics Automation*, 2011, pp. 46–51.

[26] Microsoft Robotics Developer Studio. [Online]. Available: http: //www.microsoft.com/robotics

[27] N. Ando, T. Suehiro, and T. Kotoku, "A software platform for component based RT-system development: OpenRTM-aist," in *Int. Conf. Simulation, Modeling, Programming Autonomous Robots,* New York: Springer-Verlag, 2008, pp. 87–98.

[28] C. Jang, S. Lee, S. Jung, B. Song, R. Kim, S. Kim, and C. Lee, "OPRoS: A new component-based robot software platform," *ETRI J.*, vol. 32, no. 5, pp. 646–656, 2010.

[29] G. Metta, P. Fitzpatrick, and L. Natale, "YARP: Yet another robot platform," *Int. J. Adv. Robot. Syst.*, vol. 3, no. 1, pp. 43–48, 2006.

[30] G. Sánchez and J.-C. Latombe, "A single-query bi-directional probabilistic roadmap planner with lazy collision checking," in *Proc. 10th Int. Symp. Robot. Res.*, 2001, pp. 403–417.

[31] S. Redon, A. Kheddar, and S. Coquillart, "Fast continuous collision detection between rigid bodies," *Computer Graphics Forum*, vol. 21, no. 3, pp. 279–287, 2002.

[32] K. Ahnert and M. Mulansky, "Odeint – solving ordinary differential equations in C++," in *Proc. AIP Conf. Numerical Analysis Applied Mathematics,* vol. 1389, 2011, pp. 1586–1589.

[33] E. R. Johnson and T. D. Murphey, "Scalable variational integrators for constrained mechanical systems in generalized coordinates," *IEEE Trans. Robot.*, vol. 25, no. 6, pp. 1249–1261, Dec. 2009. [Online]. Available: http://trep.sourceforge.net

[34] D. Hsu, J.-C. Latombe, and H. Kurniawati, "On the probabilistic foundations of probabilistic roadmap planning," *Int. J. Robot. Res.*, vol. 25, no. 7, pp. 627–643, 2006.

[35] N. Amato, O. Bayazit, L. Dale, C. Jones, and D. Vallejo, "OBPRM: An obstacle-based PRM for 3D workspaces," in *Robotics: Algorithmic Perspective*, P. K. Agarwal, L. E. Kavraki, and M. T. Mason, Eds. A.K. Peters, 1999, pp. 155–168.

[36] V. Boor, M. H. Overmars, and A. F. van der Stappen, "The Gaussian sampling strategy for probabilistic roadmap planners," in *Proc. 1999 IEEE Int. Conf. Robotics Automation*, pp. 1018–1023.

[37] J. Kuffner and S. M. LaValle, "RRT-Connect: An efficient approach to single-query path planning," in *Proc. 2000 IEEE Int. Conf. Robotics Automation*, San Francisco, CA, Apr. 2000, pp. 995–1001.

[38] R. Bohlin and L. E. Kavraki, "Path planning using lazy PRM," in *Proc. 2000 IEEE Int. Conf. Robotics Automation*, San Francisco, CA, 2000, pp. 521–528.

[39] E. Plaku, L. Kavraki, and M. Vardi, "Motion planning with dynamics by a synergistic combination of layers of planning," *IEEE Trans. Robot,* vol. 26, no. 3, pp. 469–482, June 2010.

[40] E. Larsen, S. Gottschalk, M. C. Lin, and D. Manocha, "Fast distance queries with rectangular swept sphere volumes," in *Proc. IEEE Int. Conf. Robotics Automation*, 2000, pp. 3719–3726.

[41] J. Pan, S. Chitta, and D. Manocha, "FCL: A general purpose library for collision and proximity queries," in *Proc. IEEE Intl. Conf. Robotics Automation*, Minneapolis, MA, May 2012.

[42] S. Chitta, I. Şucan, and S. Cousins, "Moveit!" *IEEE Robot. Autom. Mag.,* vol. 19, no. 1, pp. 18–19, Mar. 2012..

Ioan A. Şucan, Department of Computer Science, Rice University, Houston, TX. Currently with Willow Garage, Menlo Park, CA. E-mail: isucan@willowgarage.com.

Mark Moll, Department of Computer Science, Rice University, Houston, TX. E-mail: mmoll@rice.edu.

Lydia E. Kavraki, Department of Computer Science, Rice University, Houston, TX. E-mail: kavraki@rice.edu.