A Software Framework for Planning under Partial Observability

Marcus Hoerger [*], **Hanna Kurniawati** [*] **and Alberto Elfes** [§]
[*] Research School of Computer Science, Australian National University
[§] Robotics and Autonomous Systems Group, Data61, CSIRO
{marcus.hoerger, hanna.kurniawati}@uq.edu.au,
Alberto.Elfes@data61.csiro.au

Abstract

Planning under partial observability is both challenging and critical for reliable robot operation. The past decade has seen substantial advances in this domain: The mathematically principled approach for addressing such problems, namely the Partially Observable Markov Decision Process (POMDP), has started to become practical for various robotics tasks. Good approximate solutions for problems framed as POMDPs can now be computed on-line, with a few classes of problems being solved in near real-time. However, applications of these more recent advances are often hindered by the lack of easy to use software tools. Implementation of state of the art algorithms exist, but most (if not all) require the POMDP model to be hard-coded inside the program, increasing the difficulty of applying them. To alleviate this problem, we propose a software toolkit, called On-line POMDP Planning Toolkit (OPPT) (downloadable from https://github.com/rdl-algorithm/oppt). By providing a well-defined and general abstract solver API, OPPT enables the user to quickly implement new POMDP solvers. Furthermore, OPPT provides an easy-to-use plug-in architecture with interfaces to the high-fidelity simulator Gazebo that, in conjunction with user-friendly configuration files, allows users to specify POMDP models of a standard class of robot motion planning under partial observability problems with no additional coding effort.

1 Introduction

Planning under partial observability is essential to autonomous robots. To operate reliably, an autonomous robot must act strategically to accomplish its tasks, despite being subject to various motion and sensing uncertainty, and uncertainty regarding the environment the robot operates in. Due to these uncertainties, the robot does not have full observability on the state of the system. Over the past decade, substantial advances on planning under partial observability have been made. The general and mathematically principled approach for solving such problems, namely the Partially Observable Markov Decision Process (POMDP), which is notorious for its computational intractability, has started to become practical for various robotics planning problems (Pineau, Gordon, and Thrun 2003), (Kurniawati, Hsu, and Lee 2008), (Silver and Veness 2010), (Kurniawati and Yadav 2013), (Somani et al. 2013), even achieving near real-time performance for a few classes of problems (Kurniawati and Yadav 2013).

Despite these advances, an easy to use software tool for POMDP-based motion planning is lacking, which in turn hinders the community from reaping the full benefits of these new advances. Several software tools for solving POMDPs do exists, e.g., Symbolic Perseus (Poupart 2007), ZMDP (Smith 2005), APPL (of Singapore 2008), and TAPIR (Klimenko, Song, and Kurniawati 2014). To use these solvers, a user needs to first encode the POMDP model of the problem. This encoding is easy for discrete POMDP problems: Users only need to list down the values of the components that define a POMDP problem in simple file formats, such as the Cassandra file format (Cassandra 2003), PomdpX file format (Ong et al. 2010), and SPUDD format (Hoey et al. 1999). But, all software that can solve continuous POMDP problems or POMDP problems on-line require users to hard-code the problem in the software.

To alleviate the above difficulties, this paper presents On-line POMDP Planning Toolkit (OPPT) (Hoerger, Kurniawati, and Elfes 2018), a software-toolkit for approximating POMDP solutions, on-line. OPPT uses a plugin-based framework to provide flexibility and ease for users to implement new POMDP models, withouth being tied to a specific uncertainty model. For general POMDP problems, the user can implement these plug-ins. However, for standard motion planning under uncertainty problems —that is, moving from one configuration to another with errors in the effect of actions and sensing—, OPPT provides a default POMDP model, such that users only need to specify 3D models of the robot and environment, and a configuration file that specifies parameters for the probability density functions that represent uncertainties in the effect of actions, observations, and starting state, and the reward function.

OPPT allows a user to separate the POMDP model (including the robot's environment) for planning and for simulated execution. It is known that developing a faithful POMDP model is often difficult. However, it is also known that strategies computed with imperfect POMDP models can still generate relatively good robot behaviours. The ability to separate planning and execution environments will better facilitate sensitivity analysis studies of on-line POMDP

[*]CSIRO and UQ International Scholarship

solvers and allow users to better predict the performance these solvers in the physical world.

OPPT allows users to implement new POMDP solvers, too. For this purpose, OPPT provides an abstract and general POMDP solver class that is not restricted to specific data structures. Furthermore, users also have access to a rich framework that provides functionalities common for many motion planning problems, such as kinematic computations, physical simulation (via ODE (Smith 2001) in Gazebo (Koenig and Howard 2004)) of the robot and the environment it operates in and collision detection (via FCL (Pan, Chitta, and Manocha 2012)).

2 POMDP Background

Modelling a POMDP problem means defining the components of the tuple $\langle S, A, O, T, Z, R, b_0, \gamma \rangle$. The notations S, A and O are the state, action, and observation spaces. The notation T is a conditional probability function $p(s' \mid s, a)$ (where $s, s' \in S$ and $a \in A$) that represents uncertainty in the effect of actions, while Z is a conditional probability function $p(o|s, a)$ that represents uncertainty in the observations. The notation R is the reward function, which depends on the state–action pair. The notations b_0 and $\gamma \in (0, 1)$ are the initial belief and discount factor. At each time-step, a POMDP agent is in a state $s \in S$, takes an action $a \in A$, perceives an observation $o \in O$, receives a reward based on the reward function $R(s, a)$, and moves to the next state. Due to uncertainties in the results of actions and observations, the agent never knows its exact state and therefore, estimates its state as a probability distribution, called belief. The solution to the POMDP problem is an optimal policy (denoted as π^*), which is a mapping $\pi^* : \mathbb{B} \rightarrow A$ from beliefs ($\mathbb{B}$ denotes the set of all beliefs, which is called the belief space) to actions that maximizes the expected total reward the robot receives, i.e., $V^*(b_0) = \max_{a \in A} \left(R(b, a) + \gamma \int_{o \in O} p(o|b, a) V^*(\tau(b, a, o))\, do \right)$, where $R(b, a) = \int_{s \in S} R(s, a) b(s) ds$ and $\tau(b, a, o)$ computes the updated belief estimate after the robot performs action $a \in A$ from belief b and perceived $o \in O$.

3 On-line POMDP Planning Toolkit (OPPT)

OPPT separates implementations of POMDP models from solvers. To ease implementation of POMDP models, OPPT allows users to specify the state, action, and observation spaces via a configuration file, and uses a plug-in architecture to implement transition, observation, reward functions, and the initial belief. These plug-ins may have optional variables. The values of such variables are specified in the configuration file (the same file that specifies the three spaces of a POMDP problem). The details of the plug-in architecture are in Section 3.2. OPPT implements a POMDP model of a standard motion planning under partial observability problem (defined in Section 4.1), which means that for such problems, a user only needs to specify a configuration file. For more general problems, a user needs to implement the appropriate plug-in(s) as necessary and write a configuration file that specifies the state, action, and observation spaces, and plug-in options.

A new solver can be implemented via program API, as described in Section 4.4. The default solver in OPPT is Adaptive Belief Tree (ABT) (Kurniawati and Yadav 2013).

Before discussing the details of how the POMDP models and solvers are implemented in OPPT, let's first discuss its overall architecture.

3.1 Architecture Overview

The overall architecture of OPPT is shown in Figure 1. OPPT separates implementations of POMDP models and solver. Furthermore, for model implementation, OPPT allows separate models for planning and execution, although of course the same model can be used both for planning and execution. At the core of the model are the `ProblemEnvironment`, `RobotEnvironment` and `Robot` classes.

ProblemEnvironment. The `ProblemEnvironment` is the main component of OPPT. It is responsible for initializing, setting-up, and running a POMDP problem, as well as loading and initializing an instance of the POMDP solver that will be used to solve the specified POMDP problem. `ProblemEnvironment` follows the general work-flow of online POMDP planning, which includes the following high-level steps:

1. Use the `Solver` to improve the policy with respect to the current belief (`Solver::improvePolicy`).
2. Get the best action to apply to the robot (via `Solver::getNextAction`).
3. Apply the action to the robot and sample an observation and a reward.
4. Inform the solver about the action taken and observation received to update its belief (`Solver::updateBelief`).
5. Repeat steps 1–4 until a terminal state is reached, or a maximum number of steps is exceeded

After a simulation run is finished, the `ProblemEnvironment` generates an output file containing statistics about the simulation run.

RobotEnvironment. The `RobotEnvironment` class represents an interface for the `Solver` to communicate with a `Robot`. It also contains a geometric representation of the environment and the robot (the latter via the `Robot` class). At start-up, the `ProblemEnvironment` instantiates two `RobotEnvironments`: One will be used for planning, while the other for simulated execution. This class also sets up the Gazebo interface. Many robot motion planning problems involve robots with complex non-linear dynamics and observation function that cannot be modelled in closed-form. For such problems, OPPT uses the ODE physics engine (via Gazebo) to compute the robot's dynamics. The interface thereby serves two purposes: It maintains a kinematic and dynamic model of the environment and the robot, using the underlying model structure of Gazebo, and provides simple methods that allow the transition and observation plug-ins to communicate with the physics engine to simulate environment, robot and sensor dynamics.

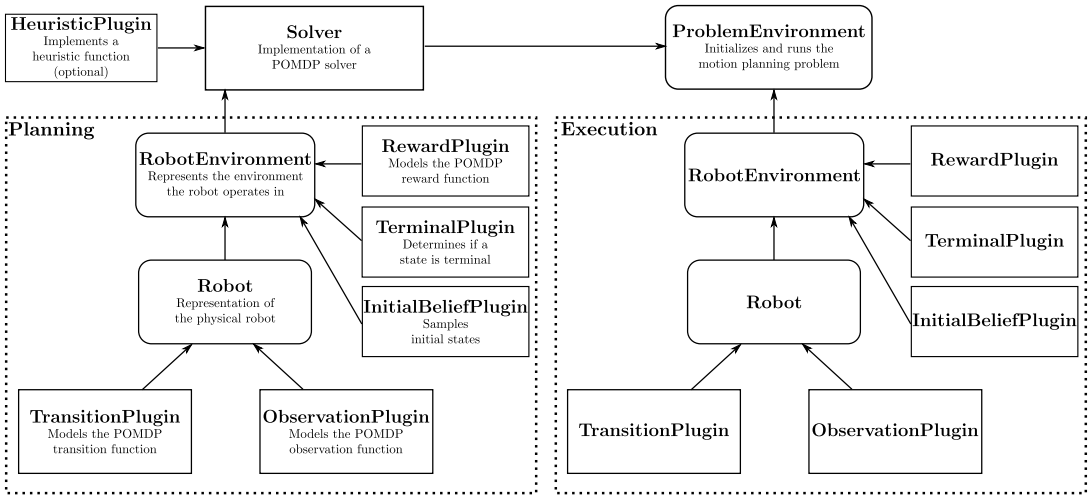


Figure 1: OPPT architecture overview. Rounded boxes represent core components of the OPPT framework. Square-shaped boxes represent components for which alternative implementations can be provided. A directed arrow from A to B depicts "A is owned by B"

`RobotEnvironment` class automatically sets-up Gazebo, based on a particular environment and robot model, and a state, action and observation description within the problem configuration file, to free the user from any additional set-up operations.

Robot. The `Robot` class is a general representation of the physical robot. It maintains the `TransitionPlugin` and the `ObservationPlugin` and provides methods (`transitionState` and `getObservation`) that allows the `Solver` to communicate with these plugins. The `Robot` class is also responsible to construct the underlying state, action and observation spaces of the robot. This is done automatically given a specific problem description. Furthermore, this class provides methods to determine if a state is valid (via the `isValid` method) or a terminal state (via the `isTerminal` method), which use the `TerminalPlugin`

3.2 Plug-In Architecture.

OPPT uses a plug-in architecture similar to the one used in Gazebo (Koenig and Howard 2004), to provide flexibility in extending both POMDP model and solver. These plug-ins are implemented as shared libraries that are loaded dynamically during runtime.

In terms of the model implementation, plug-ins are used to implement the transition, observation, and reward functions, and the initial belief (and terminal states, if required) components of a POMDP model. OPPT provides one plug-in type for each POMDP component. For each plug-in, OPPT provides an implementation for the standard motion planning problem (see Section 4.1). Standard plug-ins that specify the transition and observation functions and initial belief defines only the types of the distributions. The parameters of the distributions are provided as inputs and are specified in the configuration file. This means that to implement a POMDP model for a standard motion planning under partial observability, a user only needs to write a configuration file that specifies the state, action, and observation spaces, and the parameters of the distributions.

Furthermore, since plug-ins can be exchanged during runtime, OPPT provides better support for changing environments, commonly encountered in long-term autonomy tasks. For instance, when the quality of a sensor deteriorates, one can replace the observation function, while OPPT is running.

The default solver in OPPT is Adaptive Belief Tree (ABT) (Kurniawati and Yadav 2013). ABT is derived from Monte Carlo Tree Search, which requires a rollout strategy to estimate the values of the leaf nodes of the tree. In OPPT, this strategy is implemented as a heuristic plug-in that users can easily replace.

4 Working with OPPT

4.1 Standard Motion Planning Problems

As discussed earlier, OPPT implements the model plug-ins for a standard motion planning under partial observability problem. In this problem, one or more robots must move from an initial configuration to a configuration in a goal region. The initial configuration is not known exactly, and is represented as a uniform distribution with bounded support. The exact bounds are given as input parameters. Actions and observations are disturbed by additive Gaussian noise, whose parameters are given as input parameters. The robot and sensor dynamics are simulated by the physics engine within Gazebo. These dynamics can be linear or non-linear. Therefore, despite additive Gaussian noise in the transition and observation functions, the beliefs may not be Gaussian (even if the initial belief was Gaussian). The robot's environment is fully observable, though it may change. The reward function is designed such that the robot receives a penalty

Table 1: State and observation configuration variables for motion planning problems

jointPositions, jointVelocities	Joint angles in rad, joint velocities in rad/s
linkPoses. *linkPositionsX, linkPositionsY, linkPositionsZ,* *linkOrientationsX, linkOrientationsY, linkOrientationsZ*	Poses, positions and orientations of the local link frames w.r.t. world frame
linkVelocitiesLinear, linkVelocitiesAngular *linkVelocitiesLinearX, linkVelocitiesLinearY, linkVelocitiesLinearZ* *linkVelocitiesAngularX, linkVelocitiesAngularY, linkVelocitiesAngularZ*	Linear and angular velocities of the local link frames w.r.t. world frame
additionalDimensions	Additional state dimensions that are not considered by the physics engine

Table 2: Action configuration variables for motion planning problems

jointTorques	Input for torque controlled joints
jointPositions	Input for position controlled joints
jointVelocities	Input for velocity controlled joints
additionalDimensions	Additional action dimensions that are not considered by the physics engine

when colliding with an obstacle, a small penalty for every step it performs and a large reward when reaching the goal area. The exact penalty and reward are given as input parameters. States are terminal when the robot collides with an obstacle or reaches the goal area.

The configuration file enables the user to specify an instance of this class of standard motion planning problems. Here, the user provides state, action and observation descriptions of the POMDP model. A list of state, action and observation descriptions the user can define are shown in Table 1 and Table 2. Additionally the problem configuration file specifies the set of POMDP plugins that will be loaded during runtime, and provides a reference to the robot and environment model files. Furthermore it is possible to define additional parameters that are being used by a specific `Solver` implementation.

For motion planning problem that fits the above description, a user can use the default plug-ins, specifies the state, action, and observations spaces and the input parameters in the configuration file. The kinematic and dynamic model of the robot and the environment the robot operates in are given as inputs, and defined using the SDF format (Foundation 2014), a XML-like descriptive format that contains a precise kinematic and dynamic description of the environment and the robot. These SDF-models are used by the `GazeboInterface` to initialize the underlying physics engine. Within the SDF-model files, the user can attach sensors to the robot that are used by the standard observation plug-in.

4.2 General Planning Problem

For problems that do not fall into the class of standard motion planning problems, such as grasping, target-tracking and environmental exploration problems, the user can provide custom implementations of the plug-ins that define a particular POMDP problem. The plug-ins are designed such that only a small number of virtual methods have to be implemented. Each plug-in must implement the `load` method, which is called after a plugin has been instantiated. Here, the user can perform set-up operations for any custom data structures that are maintained within a plugin. Additionally, a pointer to the `RobotEnvironment` is passed to the `load` method that can be used by a specific plugin implementation (e.g. for collision checking).

We emphasize that OPPT does not enforce a particular uncertainty model. Instead, the user has to define how states, actions and observations are affected by the uncertainties within the `transitionState` (for the transition plugin) and `getObservation` (for the observation plugin) methods. The user can also define their own distribution representation within the plug-ins.

4.3 User Interaction

A benefit of the default solver, ABT, is that its policy can be adjusted when the environment changes. To reap this benefit, OPPT allows users to interact with the robot's operating environment using the Gazebo client GUI, during run-time. They can add and remove obstacles or change the pose of obstacles. If changes in the environment are known *a priori*, users can define them inside the configuration file. Here, the user specifies at which time step a specific change to the environment occurs. When the environment changes, the `Solver` will be informed about these changes via the `Solver::handleEnvironmentChanges` method. Implementing this method is optional, but if the user wishes to adapt the policy to the environment changes, the user must implement this method. This method has been implemented for our default solver.

For problems with fully observable environments, additional care is needed. Changes in the environment during run-time are always applied to the `RobotEnvironment` that is used by the `ProblemEnvironment` to execute a policy. However, it is possible reflect these changes in the planning environments as well.

For visualizing the motion planning progress, OPPT provides a lightweight standalone GUI that visualizes

the 3D-environment, the current state of the robot, and the current belief during run-time. It is based on RViz (Foundation 2007), a visualization toolkit within the ROS framework (Quigley et al. 2009). This GUI can also be used to replay the output files that are generated by the `ProblemEnvironment` after each simulation run.

4.4 Implementing New Solvers

In addition to reducing the difficulty of using on-line POMDP solvers, OPPT aims to ease implementation of new POMDP solvers. To this end, OPPT provides a general `Solver` interface, which is an abstract class that provides three key methods that must be implemented for new solvers.

The first method is `improvePolicy`, which is called by the `ProblemEnvironment` at each planning step. Within this method the `Solver` calculates the best policy from the current belief. Note that OPPT does not enforce a specific belief data structure. Depending on the solver, a belief can have very different representations, such as a set of particles or a multivariate-normal distribution. A `Solver` implementation therefore has to provide its own internal belief data structure.

The second key method is the `Solver::getNextAction`, which is called by the `ProblemEnvironment` after the `Solver::improvePolicy` method is finished. This method should return an action according to the calculated policy, such that this action maximises the expected discounted future reward the robot receives when executing this action.

The third key method is the `Solver::updateBelief`, which takes the action the robot has performed and the observation that has been received as input arguments. In this method, the `Solver` performs a belief update according to the action and the observation. As mentioned above, a `Solver` is not restricted to a specific type of belief, therefore the user has to implement his/her own belief update functionality. OPPT provides an implementation of the Sequential-Importance-Resampling particle filter (Arulampalam et al. 2002), which can be used when the belief is represented by a set of particles.

Apart from these three core methods, the `Solver` interface provides a set of optional methods for serialization and visualization.

5 Example Problems

OPPT is written in the C++ programming language using the C++11 specification. The source code can be downloaded from the OPPT website: https://github.com/rdl-algorithm/oppt. It implements the standard motion planning under partial observability problem, as described in Section 4.1, and ABT, a state-of-the-art online POMDP solver that can adapt its policy to changes in the POMDP model. Also included are a number of example problem scenarios: An instance of the Rocksample problem (Smith and Simmons 2004), a simple car-like robot with second-order dynamics operating inside a maze environment shown in Section 5.2, 2DOF and 4DOF-manipulators with torque control operating inside a 3D-environment populated by static obstacles, and a 7DOF Kuka IIWR robot with torque control operating inside an office environment. The following subsections describe how to implement some of these problem examples in OPPT.

5.1 2DOF Manipulator

This problem scenario, illustrated in Figure 2(a) is an example of our standard motion planning under partial observability problem. Therefore, we can specify its POMDP model without additional implementation effort. The problem consists of a simple 2DOF-manipulator problem in which the robot operates within a 3D-environment populated by a static obstacle. The robot has to move from a known initial state to a state where the end-effector lies inside a goal-area (marked by the green sphere) while avoiding collisions with the obstacle. The robot consists of two box-shaped links connected by a revolute joint. Furthermore the first link is connected to a fixed base via a revolute joint. Both joints are torque-controlled. In this problem the state of the robot consists of the angles and velocities of both joints, and the actions are the input torques for both joints. Furthermore the robot is equiped with two types of sensors: The first sensor measures the joint velocities, wherease the second sensor measures the pose of the second link inside the robot's workspace with respect to the world frame. We assume that both the actions and the observations are disturbed by zero-mean additive Gaussian noise.

The robot receives a penalty of -500 when it collides with the obstacle, and a reward of 1000 when it reaches a goal state. Additionally, to encourage the robot to move to the goal quickly, it receives a penalty of -1 for every step it takes. A simulation run ends if the robot either collides with the obstacle, reaches a goal state, or when the maximum number of 50 steps is exceeded.

For this problem, we maintain a geometric and dynamic representation of the robot in the `GazeboInterface` and use ODE physics engine (via Gazebo) to simulate the robot dynamics.

To specify the POMDP model of this problem in OPPT, we only need to set the configuration file. The state space is specified under the `[state]` section as follows:

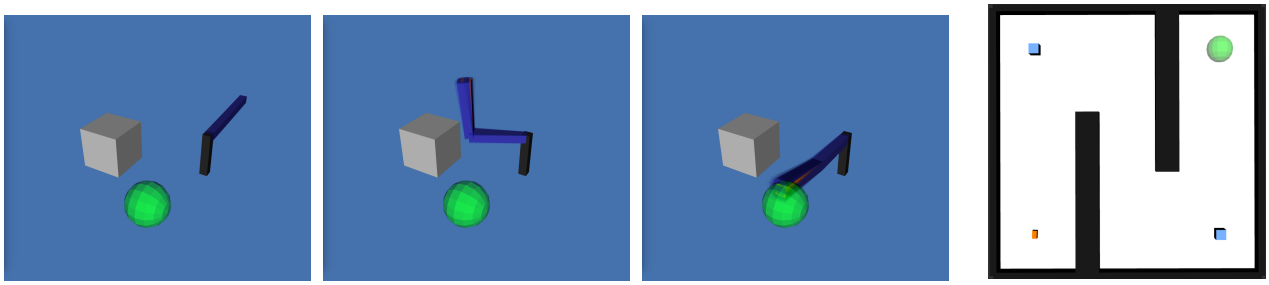
```
[state]
jointPositions = [joint1, joint2]
jointVelocities = [joint1, joint2]
```

With this description, OPPT models the states of the robot as a 4D-vector consisting of the joint angles and joint velocities. Note that the joint names have to be consistent with the ones used in the robot model file.

Next, we have to specify the POMDP actions:

```
[action]
jointTorques = [joint1, joint2]
```

This tells OPPT that the actions are 2D-vectors consisting of the joint torques. This specification also provides enough information to the `GazeboInterface`, such that an action vector is applied to the specified joints.



(a) 2DOF-manipulator (b) Car-like robot

Figure 2: Problem scenarios. (a) A 2DOF-manipulator has to reach a goal area within the environment (green sphere) while avoiding collisions with the obstacle (grey box). The three images (from left to right) illustrate a typical simulation run after $t = 0$, $t = 12$, and $t = 22$ steps. The beliefs at these steps are depicted by a set of blue particles. (b) A car-like robot (orange rectangle in the lower-left corner) drives on a flat xy-plane populated by static obstacles (black areas). The goal is to reach the area in the upper-right corner (green sphere) without collisions with any obstacles. The two blue squares in the upper-left and lower-right corners are the beacons that help localization.

Similarly, we have to specify the observations the robot can perceive:

```
[observation]
jointVelocities = [joint1, joint2]
linkPoses = [link2]
```

This specifies that the observations are 8D-vectors consisting of the joint velocities and the 3D-pose of the second link (poses are represented as 6D-vectors consisting of a position and an axis-angle orientation component).

The above specifications (in a single configuration file), together with the 3D model of the robot and environment geometry, are all that is required to specify a POMDP model when the problem is an instance of OPPT's standard motion planning under partial observability problem.

5.2 Car-like robot with 2nd-order-dynamics

For this problem, the robot and sensor dynamics are defined as closed form dynamic equations. Therefore, the transition and observation functions in the standard problem (where we rely on ODE and Gazebo simulator) must be modified. To this end, we need to provide custom implementations of the transition and observation plug-ins. Note that only these two plug-ins need to be implemented, the rest of the plug-ins can use the standard implementations. We consider a non-holonomic car-like robot that drives on a flat xy-plane inside a 3D environment populated by obstacles as shown in Figure 2(b). The robot must drive from a known start state to a position inside the goal region (marked as a green sphere) without colliding with the obstacles.

The state of the robot consists of its the position and orientation on the plane and its linear velocity expressed in the local frame of the robot. The actions are defined as the acceleration and the steering wheel angle. We assume that the robot is equiped with two types of noisy sensors: The first sensor receives signals from two beacons that are location inside the environment (blue squares), whereas the second sensor measures the linear velocity of the robot. Details on the transition and observation models used for this problem can be found in (Hoerger et al. 2016).

To specify this POMDP model, we implement the transition and observation plug-ins. The rest of the model components follows the standard problem class, as provided by OPPT, and specified in the configuration file. The state is specified by adding the link of the robot to the *linkPositionsX*, *linkPositionsY* and *linkOrientationsZ* parameters. Now, note that our options for action variables (Table 2) does not include acceleration. Therefore, for the actions of this scenario, we set *additionalDimensions=2*, where these two dimensions represent the acceleration and the steering wheel angle, which will be used in the transition plug-in implementation. Similarly, for the observations we set *additionalDimensions=3*, representing the two observed beacon signals and the linear velocity of the robot with respect to its local frame.

5.3 Rocksample

Rocksample (Smith and Simmons 2004) is a well-known scalable benchmark problem for POMDP solvers. However, this is a robot exploration problem and is outside of our standard class of motion planning under partial observability problems. Therefore, we have to provide custom implementations of each model plug-in. Rocksample models a Mars rover seeking to collect samples from valuable rocks in the environment. The rock locations are known exactly, but the quality of these rocks ("good" or "bad") is not known *a priori*. The rover's task is to sample as many "good" rock as possible in the fastest time possible. The robot is equipped with a noisy long-range sensor that can be used to check a rock. The precision of this sensor decreases exponentially as a function of the Euclidean distance of the robot to the target rock. Further details of the Rocksample problem can be found in (Smith and Simmons 2004). For this problem, we use Rocksample78, which means 8 rocks exist in a 7X7 grid cells.

To model this problem, we first specify the state, action, and observation spaces as follows:

```
[state]
linkPositionsX = [RocksampleRobotLink]
linkPositionsY = [RocksampleRobotLink]
```

```
additionalDimensions = 8
```

The first two parameters tell OPPT that part of the state variable consists of the x and y position of the robot within the environment. Since the rock states cannot be modelled according to a specific physical quantity of the robot, we need eight additional state dimensions (denoted by the *additionalDimensions* parameter) —each dimension representing the goodness of a rock.

```
[action]
additionalDimensions = 1
```

Since the provided action variables are all continuous, but this problem has a discrete action space, the action space is specified as a single variable, defined via $additionalDimensions$, that encodes the discrete action set. The transition plug-in will define the values and the meaning of each action in this set. Similarly, since the set of observations consists of two discrete observations, we set

```
[observation]
additionalDimensions = 1
```

More problem examples and their implementations are available in the release version of OPPT.

6 Performance and Scalability

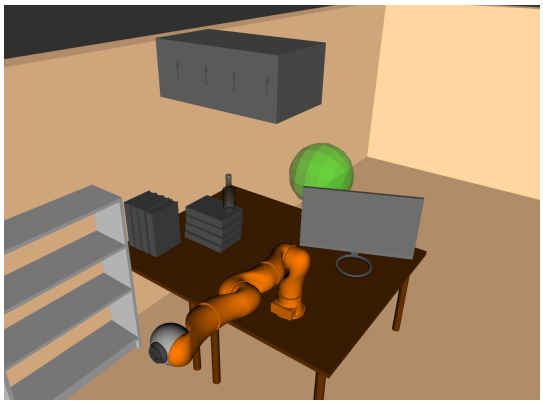


Figure 3: A 7-DOFs Kuka LBR iiwa robot operating inside an office environment. The aim is to reach the goal area (green sphere) without colliding with the interior

OPPT is designed such that the core components of the framework have little-to-no impact on the performance and scalability of new solvers. The majority of the computing cost is used by the POMDP plug-ins, the collision-detection function (for which the user can provide alternative implementations) and the solver itself. To investigate how the majority of the CPU resources are used, we ran a CPU profiler for 10 simulation runs on the Rocksample problem, the Car-like robot scenario, and a scenario of a 7-DOF Kuka arm operating in an office environment populated by complex obstacles (illustrated in Figure 3). Similar to the 2DOF Manipulator, the 7-DOF problem is defined using a problem configuration file and the standard POMDP plug-ins only, without any additional coding.

For the Rocksample problem, 85.2% of the CPU time is used by ABT to construct and maintain the belief tree, whereas 12.2% of the CPU time is used by the POMDP plug-ins to sample states and observations. The remaining CPU time is used for setup operations.

In the Car-like problem, the transition plug-in uses 60.8% of the CPU time, as it uses collision-detection, while 34.1% is used by the heuristic plug-in that is used by ABT to estimate the values of leaf nodes in the belief tree. The rest of the CPU time is used by ABT to maintain the belief tree.

For the Kuka robot, the majority of the CPU time (around 66.7%) is used by the transition plug-in, due to the physics engine and collision-detection. Around 28.2% is used by the heuristic plug-in. The rest of the CPU time is used by ABT to construct and maintain the belief tree.

In all three scenarios, methods that are specific to the OPPT framework didn't show up in the profiling statistics.

Apart from a geometric representation of the environment and the robot, OPPT doesn't maintain any internal data structures. In other words, the memory resources are used mainly by the POMDP solver, rather than the core framework.

7 Application of OPPT to a Physical Planning Problem under Uncertainty

Figure 4: Scenario for the candy-server problem

In this problem scenario we consider a manipulation task where a physical Kinova JACO manipulator interacts with the environment while being subject to significant uncertainties in motion, sensing and its understanding of the environment. Specifically, the manipulator must grasp and pick-up a cup (with known geometry) that is placed at an arbitrary position on a table in front of the robot, use the cup to scoop candies from a box of candies located next to the table, and then place the cup containing candies back on the table. The problem scenario is depicted in Figure 4. The location of the table, the location of the cup on the table, and the location of the box of candies are initially unknown and are estimated using a perception pipeline that utilizes the robot's Kinect camera. The average perception error in the object localization is 3cm. Furthermore, the cup location can change at run-time, as shown in Figure 5, e.g. a user might pick-up

(a) (b) (c) (d)

Figure 5: Snapshots of the picking task in the candy-server problem. The robot attempts to pick-up the cup (a) but the cup position is changed to the left corner of the table (b). It performs a re-localization, and attempts to pick up the cup again. But, the cup is again moved to a location close to the robot (c), resulting in the robot to attemp to re-localize the cup again. The robot then successfully picks-up the cup at its the new location (d).

Figure 6: Snapshots of the picking task in the candy-server problem with multiple number of obstacles on the table.

the cup and place it at a different location on the table while the robot is moving to pick it up, whereas the surface of the candies in the box changes after every scoop. For the picking task, we also consider a problem scenario in which the robot must avoid collisions with obstacles that are placed on the the table, as shown in Figure 6.

From a planning perspective, this is a problem of significant complexity due to its long planning horizon. We therefore divide the whole task into four sub-tasks (picking the cup, approaching the candy-box, scooping candy and placing the cup back on the table) that are being solved sequentially. Each subtasks is modelled as a POMDP with its own set of POMDP plug-ins. Once a subtask is solved, we dynamically load the POMDP plug-ins for the next subtask during run-time without having to restart the solver. For this problem we utilize OPPT's solver API to implement a multithreaded variant of the default solver ABT in which the policy computation, policy execution and belief update are running in parallel to reduce the delay between steps. Details of the system, the POMDP formulation of the subtasks and the integration of the perception pipeline into OPPT can be found in (Hoerger et al. 2019).

To test the entire planning process, from picking up an empty cup until delivering a cup of candy back to the table, we ran the entire system for 7 consecutive days in a live-demo setting. For this we let bystanders place the cup at random positions on the table ahead of every execution run. In approximately 150 runs we achieved a success rate of over 98%, i.e. the robot was able to pick up the cup, scoop candy and deliver the filled cup back to the table, showing the robustness of OPPT in solving the entire planning problem in a non-sterile setting.

8 Summary

This paper presents OPPT, an open-source software framework for on-line POMDP planning. Current software tools for on-line POMDP planning are either limited to a specific solver, or require problems to be hard-coded within the provided implementation. OPPT alleviates both limitations by providing a rich framework for the standard class of motion planning under partial observability problems and a plug-in based architecture. It provides a general API for developing on-line POMDP solvers further, and implements ABT (Kurniawati and Yadav 2013) —an on-line POMDP solver that can adapt its solutions to changes in the POMDP model— as its default solver. OPPT allows users to specify a POMDP model via plug-ins and a simple configuration files. Furthermore, for the standard class of motion planning problems, users can specify a POMDP model with no coding effort, via a configuration file and 3D models of the robot and the environment.

We hope OPPT reduces the difficulty of applying recent advances in POMDP-based planning to robotics tasks. We also hope this software will encourage and support the community to further extend the capabilities of POMDP solvers, and decision making under uncertainty in general, so as to address a major bottleneck for reliable and robust autonomous robot operations.

References

Arulampalam, M. S.; Maskell, S.; Gordon, N.; and Clapp, T. 2002. A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking. *IEEE Transactions on signal processing* 50(2):174–188.

Cassandra, A. R. 2003. Pomdp format. http://pomdp.org/code/pomdp-file-spec.html.

Foundation, O. S. R. 2007. RViz. http://wiki.ros.org/rviz.

Foundation, O. S. R. 2014. SDFormat. http://sdformat.org.

Hoerger, M.; Kurniawati, H.; Bandyopadhyay, T.; and Elfes, A. 2016. Linearization in motion planning under uncer-

tainty. In *Proceedings of the 12th International Workshop on the Algorithmic Foundations of Robotics (WAFR)*.

Hoerger, M.; Song, J.; Kurniawati, H.; and Elfes, A. 2019. Pomdp-based candy server: Lessons learned from a seven day demo. In *Proc. AAAI International Conference on Autonomous Planning and Scheduling (ICAPS)*, 698–706. AAAI.

Hoerger, M.; Kurniawati, H.; and Elfes, A. 2018. A software framework for planning under partial observability. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 7576–7582. IEEE.

Hoey, J.; St-Aubin, R.; Hu, A.; and Boutilier, C. 1999. Spudd: Stochastic planning using decision diagrams. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, 279–288. Morgan Kaufmann Publishers Inc.

Klimenko, D.; Song, J.; and Kurniawati, H. 2014. Tapir: A software toolkit for approximating and adapting pomdp solutions online. In *Proc. Australasian Conference on Robotics and Automation*.

Koenig, N., and Howard, A. 2004. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, 2149–2154. IEEE.

Kurniawati, H., and Yadav, V. 2013. An Online POMDP Solver for Uncertainty Planning in Dynamic Environment. In *ISRR*.

Kurniawati, H.; Hsu, D.; and Lee, W. 2008. SARSOP: Efficient point-based POMDP planning by approximating optimally reachable belief spaces. In *RSS*.

of Singapore, N. U. 2008. APPL. http://bigbird.comp.nus.edu.sg/pmwiki/farm/appl/.

Ong, S.; Png, S.; Hsu, D.; and Lee, W. 2010. Planning under uncertainty for robotic tasks with mixed observability. *IJRR* 29(8):1053–1068.

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

Pineau, J.; Gordon, G.; and Thrun, S. 2003. Point-based value iteration: An anytime algorithm for POMDPs. In *IJCAI*, 1025–1032.

Poupart, P. 2007. Symbolic-perseus. https://cs.uwaterloo.ca/~ppoupart/software.html#symbolic-perseus.

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

Silver, D., and Veness, J. 2010. Monte-Carlo Planning in Large POMDPs. In *NIPS*.

Smith, T., and Simmons, R. 2004. Heuristic search value iteration for POMDPs. In *UAI*.

Smith, R. 2001. Open dynamics engine. http://www.ode.org/.

Smith, T. 2005. ZMDP. http://longhorizon.org/trey/zmdp/.

Somani, A.; Ye, N.; Hsu, D.; and Lee, W. S. 2013. DESPOT: Online POMDP planning with regularization. In *NIPS*, 1772–1780.