POMDPs.jl: A Framework for
Sequential Decision Making under Uncertainty

Maxim Egorov   megorov@stanford.edu
Zachary N. Sunberg   zsunberg@stanford.edu
Edward Balaban   ebalaban@stanford.edu
Tim A. Wheeler   wheelert@stanford.edu
Jayesh K. Gupta   jkg@cs.stanford.edu
Mykel J. Kochenderfer   mykel@stanford.edu
Department of Aeronautics and Astronautics
Stanford University
Stanford, CA 94305, USA

Editor: Antti Honkela

Abstract

POMDPs.jl is an open-source framework for solving Markov decision processes (MDPs) and partially observable MDPs (POMDPs). POMDPs.jl allows users to specify sequential decision making problems with minimal effort without sacrificing the expressive nature of POMDPs, making this framework viable for both educational and research purposes. It is written in the Julia language to allow flexible prototyping and large-scale computation that leverages the high-performance nature of the language. The associated JuliaPOMDP community also provides a number of state-of-the-art MDP and POMDP solvers and a rich library of support tools to help with implementing new solvers and evaluating the solution results. The most recent version of POMDPs.jl, the related packages, and documentation can be found at `https://github.com/JuliaPOMDP/POMDPs.jl`.

Keywords: POMDP, MDP, sequential decision making, Julia, open-source

1. Introduction

Recent advances in algorithms and hardware make partially observable Markov decision processes (POMDPs) a viable model for a variety of applications ranging from aircraft collision avoidance (Wolf and Kochenderfer, 2011) to spoken dialog systems (Young et al., 2013). In general, POMDPs can be applied to a wide variety of problems that include state and dynamic uncertainties as well as continuous and discrete domains (Kochenderfer, 2015). State-of-the-art algorithms like POMCP (Silver and Veness, 2010) and MCVI (Bai et al., 2010) can solve problems with millions of states and with continuous state spaces. However, working with POMDPs is challenging due to their probabilistic nature and the complexity of the algorithms used to solve them.

POMDPs.jl is an interface for solving POMDPs written in the Julia programming language (Bezanson et al., 2012). It is designed to support users in three different roles: 1) defining problems, 2) creating solvers, and 3) running experiments. The goals of POMDPs.jl are to allow users in these roles to easily build on code written by others through a single
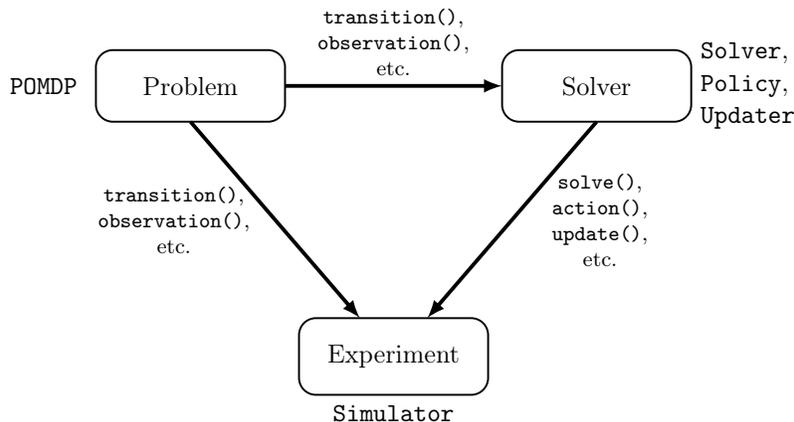
Figure 1: POMDPs.jl framework architecture. The three main concepts in the architecture: problem, solver and experiment are supported by abstract types in the framework as well as necessary function implementations.

unified interface, simplify the process of benchmarking algorithms against one another, and encourage growth of the software ecosystem through open-source contributions. This ecosystem is organized as an online community, JuliaPOMDP. The centerpiece of the community is the POMDPs.jl package — a unified, implementation-free interface providing access to a growing collection of state-of-the-art solvers, benchmark problems, and libraries of support tools. An outline of the POMDPs.jl architecture and its core concepts are shown in Fig. 1. POMDPs.jl provides abstract types corresponding to each of the core concepts and a list of functions that defines a standard interface. Users implement concrete types that represent problems or solvers and specify their behavior by defining methods.

2. Existing Frameworks and Related Work

There are a number of frameworks that are capable of solving decision making problems in fully observable settings, but few accommodate partially observable domains. The most closely related frameworks to POMDPs.jl are APPL (Kurniawati et al.), AI-Toolbox (Bargiacchi), and ZMDP (Smith) in that they can handle problems with partial observability. However, these frameworks either support only a generative model problem definition (APPL for MCVI) or explicit problem definitions using probability tables (APPL for SARSOP, AI-Toolbox and ZMDP), but not both. This limits the flexibility of the frameworks to handle both discrete and continuous domains. While the fact that these frameworks are all written in C++ leads to performant code, it makes prototyping slow and challenging for new users. It is also comparatively more difficult to extend these frameworks to new domains and implement new algorithms due to their complicated hierarchies and lack of modularity in the source code. A number of reinforcement learning frameworks exist that are designed to be easily extendable such as BURLAP (Diuk et al., 2008), RLPy (Geramifard et al., 2015), and rllab (Duan et al., 2016). However, these frameworks primarily cater to problems that are fully observable.

3. Why POMDPs.jl?

POMDPs.jl aims to simplify the task of writing problems, implementing solvers, and running experiments in the context of POMDPs. The advantages of POMDPs.jl are as follows:

Simplicity: The POMDPs.jl interface provides a minimal set of types and functions necessary to define a problem, a solver, and experiments in a partially observable setting. The POMDPs.jl interface contains 10 abstract types and 37 functions, of which, each user will define a subset for their problem. For example, if a user wants to solve an MDP rather than a POMDP, they do not need to implement functions involving observations. Examples of function requirements for the MCTS and SARSOP solvers are shown in Table 1.

Table 1: A section of the function table for two solvers that use POMDPs.jl

functions	states	actions	observations	transition	observation	reward	⋯
MCTS	n/a	✓	n/a	✓	n/a	✓	⋯
SARSOP	✓	✓	✓	✓	✓	✓	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Expressiveness: The POMDPs.jl interface provides flexibility to handle problems that are fully or partially observable, problems that are continuous or discrete, and problems with infinite and finite horizons. Solvers may leverage explicitly-defined distributions, if they are available, or only use samples from a generative model. Moreover, all POMDPs.jl problems can be defined in code. Julia makes it easy to prototype new code, removing the need for problem specification or solver configuration files written in another format.

Extensibility: The interface allows new algorithms to be implemented with minimal effort. By providing an expressive interface to a POMDP model, POMDPs.jl allows algorithm writers to access any component of the model with simple function calls. Since all solvers are implemented within a unified framework, POMDPs.jl also allows for standardized benchmarking across algorithms. It is also simple to call programs written in other languages from Julia, such as Python and C, allowing existing solvers to be wrapped using POMDPs.jl.

Usability: A number of ready-to-use solvers supporting the POMDPs.jl interface are available from the JuliaPOMDP community. To install the SARSOP solver, for example, the user only needs to run `POMDPs.add("SARSOP")`, and the package manager will download and compile any dependencies on all three major platforms — Linux, Windows, and OS X.

4. Concepts and Use Case Examples

This section describes the three main concepts outlined in Fig. 1, namely the problem, the solver, and the experiment. The POMDPs.jl interface consists of several abstract types and a set of functions that support these concepts.

Problem: The problem defines the MDP or POMDP model to be solved. Below is a definition for the Tiger POMDP (Kaelbling et al., 1998).

```
immutable TigerPOMDP <: POMDP{Bool, Int64, Bool}
    p_correct::Float64      # probability of hearing the tiger correctly
    discount::Float64       # discount factor
end
```

The `TigerPOMDP` type inherits from the abstract `POMDP` type (part of the POMDPs.jl interface), which is parametrized by the state, action, and observation types. In this case, the state is represented by the native `Bool` type, but, in general, these types may be user-defined. The same flexibility is available for the other components of the problem, giving the user the flexibility to define their problem in the form of their choosing.

Solver: A solver implementation typically requires three type definitions: a `Solver` that contains the parameters that define solver behavior, a `Policy` that defines a mapping from beliefs to actions, and an `Updater` that defines how the belief is updated with new observations. An example of a QMDP (Littman et al., 1995) policy type is shown below.

```julia
type QMDPPolicy{Action} <: Policy
    alphas::Matrix{Float64}     # policy alpha vectors
    action_map::Vector{Action} # indices to actions
    pomdp::POMDP                # POMDP model
end
```

A solver implementation usually includes three top-level functions: `solve`, that creates a policy for the problem given the solver, `action`, that emits an action for a belief based on the policy, and `update`, that updates the belief based on the action taken and the observation received, given the updater.

Experiment: Experiments combine problems and solvers to evaluate the quality of a solver policy. For example, the experimenter might create a simulator type, `Sim`, and a corresponding `simulate` function (an important part of the main loop is shown below).

```julia
function simulate(sim::Sim, pomdp, policy, updater, initial_dist)
    ...
    for t in 1:sim.max_steps
        a = action(policy, b)
        sp = rand(sim.rng, transition(pomdp, s, a))
        r_total += discount(pomdp)^t*reward(pomdp, s, a, sp)
        o = rand(sim.rng, observation(pomdp, s, a, sp))
        b = update(updater, b, a, o)
        ...
```

A simulation can then be run as follows:

```julia
using QMDP, POMDPModels, POMDPToolbox # import JuliaPOMDP packages
pomdp = TigerPOMDP()                   # initialize the tiger problem
solver = QMDPSolver()                  # initialize QMDP solver
policy = solve(solver, pomdp)          # compute a policy
r = simulate(Sim(), pomdp, policy, updater(policy), DiscreteBelief(2))
```

5. Conclusion

POMDPs.jl is a high level interface for working with POMDPs that allows users to easily define their problems, create new solvers, and run experiments. This manuscript provides a brief overview of the framework and its features; extensive documentation and examples for this software package can be found at `https://github.com/JuliaPOMDP/POMDPs.jl`. Future work includes integrating Julia's shared memory parallelism into sampling based solvers to improve computational efficiency, as well as introducing support for reinforcement learning algorithms.

References

Haoyu Bai, David Hsu, Wee Sun Lee, and Vien A Ngo. Monte Carlo value iteration for continuous-state POMDPs. In *Algorithmic Foundations of Robotics IX*, pages 175–191. Springer, 2010.

Eugenio Bargiacchi. AI-Toolbox. `https://github.com/Svalorzen/AI-Toolbox`.

Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *arXiv:1209.5145*, 2012.

Carlos Diuk, Andre Cohen, and Michael L Littman. An object-oriented representation for efficient reinforcement learning. In *International Conference on Machine Learning (ICML)*, pages 240–247. ACM, 2008.

Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. *arXiv:1604.06778*, 2016.

Alborz Geramifard, Christoph Dann, Robert H Klein, William Dabney, and Jonathan P How. RLPy: A value-function-based reinforcement learning framework for education and research. *Journal of Machine Learning Research*, 16:1573–1578, 2015.

Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1):99–134, 1998.

Mykel J. Kochenderfer. *Decision Making Under Uncertainty: Theory and Application*. MIT Press, 2015.

Hanna Kurniawati, David Hsu, Wee Sun Lee, and Haoyu Bai. Approximate POMDP Planning Toolkit. `http://bigbird.comp.nus.edu.sg/pmwiki/farm/appl/`.

Michael L. Littman, Anthony R. Cassandra, and Leslie Pack Kaelbling. Learning policies for partially observable environments: scaling up. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 362–371, Tahoe City, California, 1995.

David Silver and Joel Veness. Monte-Carlo planning in large POMDPs. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2164–2172, 2010.

Trey Smith. ZMDP software for POMDP planning. `https://github.com/trey0/zmdp`.

Travis B. Wolf and Mykel J. Kochenderfer. Aircraft collision avoidance using Monte Carlo real-time belief space search. *Journal of Intelligent and Robotic Systems*, 64(2):277–298, 2011.

Steve Young, Milica Gasic, Blaise Thomson, and Jason Williams. POMDP-based statistical spoken dialogue systems: a review. *Proceedings of the IEEE*, PP(99):1–20, 2013.