Kaan Ozulkulu

**Analysis of Leveraging C++ via Pybind11 for POMDP Problems Defined in Python**

In this paper, I will give a brief introduction to the Markov Decision Process (MDP) and the Partially Observable Markov Decision Process (POMDP) [1]. I will discuss their use in modeling certain problems and their importance in solving these problems. I will then introduce and walkthrough a method which I have worked on with Semanti Basu. This method, by using the Pybind11 library, enables anyone to easily model their POMDP problem in Python and solve it in a fast way without having to write a solver. Such a method is possible by binding the problem definition in Python with the solver implemented in C++, all the while giving the user control over the solver if desired. The full code can be viewed here: https://github.com/kaanozulkulu/pybind11-pomdp. I will also highlight the advantages and disadvantages of this method based on my observations and indicate how this method could be improved in the future.

A Markov Decision Process (MDP) models sequential decision making problems. If a decision making problem consists of a set of states, actions, transition probabilities, and a reward model, we can frame this problem as an MDP. Using this type of modeling, we can generalize various situations from figuring out how agents should behave in a game at each step to how a robot should move while trying to navigate its way. We can even see similarities to how we make decisions in our own lives. We assess the current situation (the state) that we are in, try to look ahead to see what action will lead us where (the transition probabilities and expected rewards) and then we try to choose the best action.

However, there is still a big gap between modeling real life problems as MDPs and how real life actually works. We make a lot of assumptions while making decisions and rely on our previous experiences or beliefs when assessing our situation and possible outcomes. Robots that are trying to navigate and make decisions in unknown territories rely on their sensors like their camera or radar while assessing their surroundings. Yet, there is always noise involved so they cannot rely fully on their sensors as well. A framework that bridges this gap and makes MDP’s more robust to uncertainty is Partially Observable Markov Decision Process (POMDP). In POMDPs, as the name suggests, the agent making decisions does not have complete knowledge of the states. Therefore, a belief state is introduced to the model that represents the probability distribution over all the possible states. In this model, as the agent cannot unequivocally know in which state it is in, the agent makes observations and updates its belief state accordingly. This is extremely helpful when trying to have robots navigate or do tasks in the real world where there is uncertainty.

Another critical area for POMDP’s real life application is a robot’s decision-making time. A popular approach to solving POMDPs is Partially Observable Monte-Carlo Planning (POMDP)
This algorithm finds the best action to take given a belief state. It simulates the outcomes of each action until the simulation reaches a terminal state or until a pre-configured limit. If there are numerous possible states or actions available to the agent, these simulations take longer to run to give satisfactory results. However, if a robot is making decisions in real life conditions, the planning should be fast. Therefore, speeding up this process as much as possible is crucial.

One aspect to tackle this problem and introduce a speed-up is to use a compiled language. However, while compiled languages like C++ provide better performance, Python is overall more popular among programmers. This could be because of Python’s simplicity and ease of use. Boston Dynamics’s Spot robot also uses Python for its SDK. Although there is also a C++ version, it is still in beta and has a smaller community. Therefore, choosing to write a POMDP library with a compiled language instead of Python might limit its accessibility and usability even if it means it would be faster. In addition to worse performance compared to compiled languages, Python also has a Global Interpreter Lock (GIL) which prevents tasks in the same process from running in parallel. With languages like C++ we do not have this issue and can utilize multithreading to speed up the completion of tasks.

Therefore, coming up with an approach where people can interact with Python code but still benefit from a compiled language’s performance has been an area of exploration in planning problems. For example the AI-Toolbox [3] uses Boost.Python [4] to have a library of reinforcement learning and planning algorithms defined in C++ and also offers a Python interface. The pomdp.py [5] library also utilizes Cython [6] to speed up solving POMDP problems defined in Python. Pybind11 [7], on the other hand, was not utilized in this area. It is a lightweight header library built with the hopes of modernizing Boost.Python and it can be used to bind C++ with Python in an intuitive manner. It also works better with modern compilers compared to Boost.Python. While Cython also provides a major performance boost by enabling type declarations and function calls in C, it is its own language, not a library for doing bindings. Therefore, we decided to utilize Pybind11 to decouple the solver and the definition of POMDP problems. This enabled us to have the codebase consist of two popular languages, Python and C++, and have them work together through a lightweight header library in an intuitive manner. This provides the opportunity to use C++ as an efficient solver and Python to define the POMDP problems. Additionally, this type of an approach can also make it easy to customize the C++ side. For example, one can call a function in Python, pass the execution of the task to C++, and utilize multiple threads to compute it. Once the threads are done the result can be returned back to Python.

Below are our results from running the same task in pure Python, Python and C++ via Pybind11 with a single thread, and Python and C++ via Pybind11 with 6 threads.

---

1 According to PYPL Popularity of Programming Language Index https://pypl.github.io/PYPL.html
2 Link to Python SDK: https://github.com/boston-dynamics/spot-sdk
3 Link to C++ SDK: https://github.com/boston-dynamics/spot-cpp-sdk
Our design of a POMDP problem in terms of how it is modeled and solved follows the "design philosophy" from Kaiyu Zheng and Stefanie Tellex’s *pomdp.py: A Framework to Build and Solve POMDP Problems* [5]. Extending *pomdp.py*’s POMDP problem design, we have initialized the components that are needed to model a POMDP as virtual classes in C++. These components such as state, action, transition model that we have mentioned earlier, are used in our planner while doing simulations. Now, when we want to define a new POMDP problem in Python, we simply need to inherit these virtual classes and implement their respective methods in Python. After accomplishing this via Pybind11, we can solve the problem that we have defined. Our current codebase demonstrates how a domain agnostic solver can be implemented and utilized by having the same solver solve two different problems without any modification. For example, the Tiger problem [5] and the RockSample problem [5] can both be solved by just running their relative Python scripts, even though the State component in the RockSample problem has different properties such as rock type and rock position. To execute this, we added these properties to the initialization of the State class in C++ and provided default values during the binding. Therefore, even if our Python definition does not have these properties in the implementation we can still use the same solver, which is the case when solving the Tiger problem.

The below sequence walks through code samples for the above process.

---

4https://h2r.github.io/pomdp-py/html/examples.tiger.html

5https://h2r.github.io/pomdp-py/html/problems/pomdp_problems.rocksample.html
1) Instantiate virtual classes in C++

```cpp
class State {
public:
    State(string name_, std::tuple<int, int> position_ = std::make_tuple(0, 0), std::vector<std::string> rocktypes_ = std::vector<std::string>(), position(position_), rocktypes(rocktypes_), terminal(terminal)) : name(name_), position(position_), rocktypes(rocktypes_), terminal(terminal) {}
    State() {}
    virtual string getname() { return name; }
    string name;
    std::tuple<int, int> position;
    std::vector<std::string> rocktypes;
    bool terminal;
    virtual ~State() {}
};
```

Listing 1: Above is an example of the virtual State class in C++ that is used in the RockSample problem. As our C++ code is meant to be domain agnostic, we add the State properties unique to the problem to the C++ side with default values. Therefore, we can still use the solver for other problems that do not have these properties.

2) Write the wrapper classes for the virtual classes. These are used for binding them with Python.

```cpp
class PyState : public py::wrapper<State> {
public:
    using py::wrapper<State>::wrapper;
    std::string getname() override {
        PYBIND11_OVERLOAD(
            std::string, /* Return type */
            State, /* Parent class */
            getname); /* Name of function in C++ (must match Python name) */
    }
};
```

Listing 2: Any method that will be used on the Python side should be added here. For the State class getname() is added.
3) Bind the C++ classes with Python

```
PYBIND11_MODULE(example, m)
{
py::class_<State, PyState, std::shared_ptr<State>> state(m,
    "State", py::dynamic_attr());
state
    .def(py::init<string, std::tuple<int, int>, std::vector<std::string>, bool>(),
        py::arg("name"),
        py::arg("position") = std::make_tuple(0, 0),
        py::arg("rocktypes") = std::vector<std::string>(),
        py::arg("terminal") = false)
    .def(py::init<>())
    .def("getname", &State::getname)
    .def_readwrite("name", &State::name)
    .def_readwrite("position", &State::position)
    .def_readwrite("rocktypes", &State::rocktypes)
    .def_readwrite("terminal", &State::terminal);
}
```
Listing 3: The virtual class, the wrapper class and a shared pointer to the class are needed. The type of State needs to be shared_ptr on the C++ side in order to refer to the binded State class. Any property of the class that will be manipulated on the Python side should be added with .def_readwrite().

4) Inherit virtual classes in Python

```
from example import State
class RockState(State):
    def __init__(self, position, rocktypes, terminal=False):
        State.__init__(self, str(position) + "|" + str(rocktypes) + "|" + str(terminal), tuple(position), tuple(rocktypes), terminal)
        self.name = str(position) + "|" + str(rocktypes) + "|" + str(terminal)
        self.position = tuple(position)
        self.rocktypes = tuple(rocktypes)
        self.terminal = terminal
```
Listing 4: The State class is now ready to be used in both C++ and Python.
This type of a sequence can be used for any new POMDP problem. In fact, one can define a whole library of problems and add properties specific to each problem to the C++ virtual classes (Listing 1), bind them with Python with default values (Listing 3) and once the C++ code is compiled all the problems in the library can be solved by just running their respective Python scripts. In our code there is currently support for a belief state that is represented as a histogram and also a particle-based representation. One can use any of the two depending on the problem or one can define their own structure for a belief state as well.

Additionally, we also ran the simulation code in a multithreaded manner. However, contrary to what was expected this did not provide an improvement in performance. This was due to the number of callbacks to the Python side during simulation. Whether it is fetching the reward value of taking an action or calling a transition function defined in Python, the threads have to wait for the callbacks to be completed. This prevented us from observing any speed-up. Yet, if there are additional calculations or helper functions in C++ that do not use any shared instances with Python we can still benefit from multithreading. As these tasks do not involve any Python callbacks, running them in parallel will improve the performance. We were able to test this by introducing a dummy task inside the simulation function in C++ and observed that running the planner on six threads was two times faster than running it on a single thread. This shows that by getting creative we can still benefit from architectures that utilize multiple threads.

An issue one could face with Pybind11 is that as the amount of callbacks between C++ and Python increase, the runtime can also increase. In the below code boxes we see two approaches of getting the next state, observation and reward during simulation. The second approach that has a single callback gives a 17% improvement in performance.

1) Three Callbacks

```cpp
std::shared_ptr<State> next_st = T->sample(state.get(), a.get());

double reward = R->sample(state.get(), a.get(), next_st.get());
std::shared_ptr<Observation> obs = O->sample(next_st.get(), a.get());
tuple<std::shared_ptr<State>, std::shared_ptr<Observation>,
      double, int> res(next_st, obs, reward, nsteps);
return res;
```

Listing 5: Here we have separate calls from C++ to the Transition model (T), Reward model (R) and Observation model (O).
2) Single Callback

```cpp
C++

tuple<std::shared_ptr<State>, std::shared_ptr<Observation>, double> tups;
tups = T->full_sample(state.get(), a.get());
tuple<std::shared_ptr<State>, std::shared_ptr<Observation>, double, int> res(get<0>(tups), get<1>(tups), get<2>(tups), nsteps);
return res;
```

```python
Python
class RSTransitionModel(TransitionModel):
def full_sample(self, state, action):
    next_state = self.sample(state, action)
    obs = O.sample(next_state, action)
    reward = R.sample(state, action, next_state)
    return (tuple([next_state, obs, reward]))
```

Listing 6: Here we have a single call from C++ to the Transition model. The Transition model calls the Observation and the Reward model inside Python and returns all the values to C++.

The 17% increase in speed with the second approach may seem counterintuitive to the whole process of having most of the planning work be done in C++ as we have now increased the work done on the Python side. By reducing callbacks we cut the time spent on fetching shared objects between C++ and Python, improving our overall performance. However, it is important to note that while the number of callbacks is something to be aware of, the amount of work that is done in Python should be kept at minimum as well. Our initial implementation of a domain agnostic C++ solver did not have any problem specific properties initialized on the C++ side. The base model only took in a string property called “name” for the State class. Therefore, we passed every required property from the Python side as a combined string. To accommodate for this the Python implementation also needed to change. For example, the position of the rocks and rock types in the RockSample problem were extracted via string manipulating helper functions in the Python side. This type of implementation made our runtime four times slower, therefore we decided to go with the option where the specific properties are added to the C++ side. These examples highlight how important it is to construct a model using Pybind11 in an intricate manner as there is a tradeoff that needs to be balanced between limiting the number of callbacks to Python by potentially doing more work in one callback, while also having minimum code in Python. Therefore, the future work to improve the current implementation could go deeper into how to intricately solve this tradeoff issue.
A more general tradeoff is between having a generalized solver in C++ and a fast one. By adding more specificity to the C++ components, we were able to have a faster solver. However, this does require some interaction with the C++ code by a new user to bind their new problems, even if it is minimal. To completely isolate C++, helper functions are required on the Python side which adds to the runtime. For example, in the RockSample problem different type of actions are differentiated via helper functions on the Python side as we have only one Action class in C++. One can also make improvements on the multithreading part. If the Python callbacks can somehow be isolated, the full potential of multithreading can be utilized with Pybind11. Finally, as these results and observations are gathered from simulation, any future work that applies this to robots will get valuable insight into how a method for solving POMDP problems using Pybind11 could be improved and, hopefully, advance current approaches that are used in robots.
References


