Snake Eyes: Decision Making Under Uncertainty in Python

Paul Motter

***Abstract*— While Julia is well suited to reinforcement learning-adjacent tasks and mathematical programming in general, it lacks the popular familiarity of the most well-known languages. I present a proof-of-concept Python package that allows the Julia infrastructure for the Decision Making under Uncertainty coursework to be interacted with from Python. The purpose of this interface is to allow students who come from a "computer science education" background—where Python is a much more common language to encouter—and are more familiar with Python to interact with the material of this class in a more comfortable way. Ideally, this would save the student some valuable time by negating the necessity of learning the idiosyncrasies of Julia. However, I find that it may be more time efficient to gain an appreciation of Julia and its efficiencies, as a fairly basic proof-of-concept Python to Julia conversion is not as performant or straightforward as one may hope.**

I. Introduction

Julia is fast, it is well designed, and it is still growing in popularity. According to one source this growth is down to its speed, its usability, and its accessability, which combine to give Julia enough popular appeal to crack the top twenty most popular programming languages worldwide [1]. These qualities give the aspiring data scientist, mathematician, or researcher plenty of reason to learn Julia, even if it is not required for coursework.

Even if Julia is growing, it has a long way to go before it catches on to the same degree as a language the Julia novice may find to feel very similar. Python is still king in the data science and academic world. It has vast support through the myriad of tools available to aid Python development, the large user base, and the great quantity of libraries available for any manner of task. In this computer science graduate student's experience, Python has been a part of a great portion of computer science classwork at several universities. Facing the daunting task of having once again go to Google for very basic syntax hints as in the formative years of such an education (what does a Julia lambda look like again?), many students may balk at a requirement to use Julia for classwork. A few weeks into the course, as they find that many of the handy IDE tools they've grown used to in languages like Python are not available for Julia, a sense of dread sets in. Students decry Julia's foibles on class project leaderboards and Piazza as once again, the prospect of another sleepless night spent trying to accomplish a task they could very easily do and have done times before in Python proves frustratingly difficult in Julia. "The basic syntax is so similar!" come the lamenting cries, "Why must Julia insist on multiple dispatch and the abandonment of dot-oriented programming when it is so dissimilar to what we are used to?"

This project attempts to answer the question of whether the coursework for Decision Making Under Uncertainty could feasibly be done in Python without necessitating a complete re-write of years worth of Julia code for DMU tasks. Essentially, could we have the best of both worlds: the disgruntled students get to use their language of choice while the course instruction team can use their proven codebase. In this report, I introduce DMUpy, a Python wrapper for the content of the DMUStudent.jl homework functionality. First, I will introduce PythonCall/JuliaCall and my decision to use it versus alternatives. I will move on to a discussion of the methods used and the decision making behind certain design choices made for the python wrapper code for each homework assignment. Finally, I will go over some performance findings before making a judgement on feasibility.

It is important to note that this project is a proof-of-concept and is not ready for the big-time, i.e. ready to be thrown into the grubby hands of the general graduate student. If the course staff chooses to go forward with the principles exhibited here, they must be ready to apply their superior knowledge of the underlying Julia and task structures to catch any logical or efficiency flaws in my implementation. For the homeworks I did attempt to create an interface for (I overlooked one entirely, and a few are incomplete—more on that later), I exposed the interfaces I used when coding my own Julia solutions which were by-and-large not the most inventive or efficient solutions possible. I also can only be called experienced in Python in comparison to my experience in Julia: I am professionally a Java/C++ programmer. Therefore, there may be some performance to be unlocked in my proof-of-concept that I missed but will be obvious to the trained eye.

II. Python/Julia Interoperability

Python is an interpreted language; Julia is compiled. Therefore, the obvious way to undergo such a project would be to call Python from Julia, where Julia's precompilation optimizations would be best leveraged and the overall task would be more performant. However, the purpose of this project is to allow the Julia-phopic student to avoid Julia to the greatest degree while allowing the greatest leverage of development tools available to Python users (for example, the notebook implementation for Python is vastly superior to that for Julia, *especially* debugging in notebooks). Therefore (and most definitely not because I got much too far down the rabbit hole of doing it this way to have enough time to switch to a Python-called-by-Julia concept), the main "working environment" for this project is Python, with Julia being accessed as necessary.

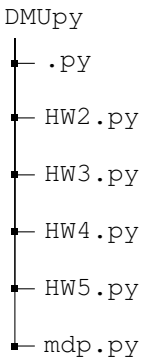

Fig. 1: The directory structure of DMUpy. Intended as a lightweight wrapper of functionality implemented in Julia, it is a very simple structure, but should be easy to maintain.

```python
import numpy as np

# rigamarole to get DMUpy package
import os
import sys
sys.path.append(os.path.abspath('..'))
from DMUpy import HW2

m = HW2.unresponsive_ACAS_MDP(10,
                      sparse_transition=
                      True)

g = HW2.grid_world(sparse_transition=True
                      )
```

Fig. 2: A code snippet from my Python solution to HW2. As you can see, there is no interaction with Julia in the loading or evaluation process.

The two most developed resources for Python/Julia Interoperability are PyCall/PyJulia [2] and PythonCall/JuliaCall [3]. Both of these systems accomplish largely the same tasks in very similar ways, however there are some differences. PythonCall/Juliacall supports more language-to-language conversions out of the box, has a more efficient wrapping system on the PythonCall side (though that is largely irrelevant here), and a similar API on both sides of the language barrier (which is very relevant to this project). JuliaCall is also more lightweight than PyJulia. Finally, there seems to be more help resources available (forum posts, etc.) for PythonCall/JuliaCall than for PyCall/PyJulia, which is a great resource when diving head-first into accessing Julia from Python.

III. Methodology

A. Basic Structure and Concept

The basic structure of DMUpy, my solution for wrapping the DMUStudent.jl homework package, is one file per homework assignment that wraps and converts the necessary objects and functions for each homework assignment (see figure 1). Each HW*.py module can be imported into a python script without any explicit interaction with Julia, and the student can complete an assignment from data initialization to evaluation completely within Python as in figure 2.

Within each HW*.py module, the relevant (PO)MDP fields and functionality are loaded into Python as appropriate. For example, the state and action lists are loaded into python completely for ease of use and calculation with Python libraries such as numpy or scipy. Functionality such as generative transitions for homeworks three through five is wrapped, where a numpy array for the states and/or actions as appropriate are passed into a wrapper function that converts the inputs to the expected Julia types, calls the relevant Julia function, then returns the output back to the Python calling environment as python types (with the notable exception of symbols).

The four homeworks present in the DMUpy package can be split into two groups: the POMDPs.jl MDP group (homeworks 2 and 3) and the CommonRlInterface.jl group (homeworks 4 and 5). For the MDP group, the file mdp.py provides a Python Mdp class that handles the basic functions of a POMDPs.jl MDP. The files HW2.py and HW3.py extend the Mdp class for problem-specific fields and functions and provide examples on how to expose additional functionality for these sorts of MDP. For the CommonRlInterface.jl group, HW4.py and HW5.py convert the relevant POMDP to a Python gymnasium environment. This has special utility for HW5, where the standardized environment will allow students to experiment with the myriad of available Python solvers.

B. Persistent Challenges

Most symbols from DMUStudent structures and functions are returned from DMUpy as a Julia symbol type. To my knowledge, there is no Python analogue to Julia's symbol type and the Julia symbols function perfectly well as Python dictionary keys, etc. It turns out that juliacall.convert() will provide a Julia symbol when passed the Julia Symbol type and a Python string, but I discovered this relatively late in the process of develop this (there is no documentation alluding to this functionality). One very annoying facet of numpy arrays is that they are not hashable, and therefore are not usable as keys to Python dictionaries, which are a hashmap implementation. This caused issues for my work with HW3 and HW4, where the methods as taught in class (and therefore most likely to be implemented by students) rely on dictionaries keyed on states. I was able to work around this by converting the numpy arrays to tuples before using them as keys to the various dictionaries, but at a performance penalty. I also encountered frequent performance issues when calling Python functions from Julia, which I will discuss in the performance section.

C. Homework 2: Value Iteration

Homework 2's basic purpose is to introduce students to one of the most fundamental concepts in Decision Making under Uncertainty: value iteration [4]. This is complicated, both from the student's point of view and from the point of view of someone trying to translate the HW2 environment to Python, of potentially very large transition matrices. The DMUStudent.HW2 environment provides a solution in the

```
pyT = {}
for key in juliaT.keys():
    value = juliaT.get(key)
    shape = (value.m, value.n)
    cols = np.array(value.colptr, dtype=
                    int) - 1
    rows = np.array(value.rowval, dtype=
                    int) - 1
    data = np.array(value.nzval)
    pyT[key] = csc_array((data, rows,
                    cols), shape=
                    shape)
```

Fig. 3: Converting a Julia sparse matrix to a scipy representation

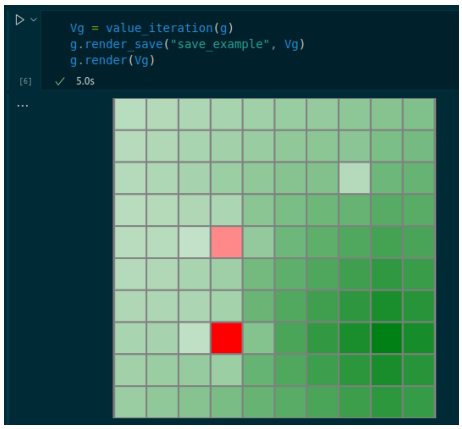

Fig. 4: Rendering the HW2.gw environment in a Python notebook

form of a CSC sparse matrix [5]. As flexible as PythonCall/JuliaCall is, there is no automatic conversion between a Julia sparse matrix and one in Python—in fact, numpy, the default conversion target for Julia structures, does not even have a sparse matrix implementation! Thankfully, the scipy package does, and it even uses the same storage format as the Julia version. It was a trivial matter to offload the members of the Julia sparse matrix into the scipy CSC sparse matrix implementation as in figure 3. Scipy CSC matricies interact well with numpy vectors, so the calculation was trivial as well.

For the gird world environment, I was able to render an image of the state colored by the value function as in Julia inside a python notebook (figure 4).

Calculation and evaluation of a value iteration solution to this problem in Python was quite performant, roughly on par with my Julia implementation of value iteration which I modeled my Python solution after. However, loading the 10 by 10 ACAS MDP took a very long time, on the order of minutes. I believe that this is due to the implementation of the ACAS MDP Python class in HW2.py, which loads the states, actions, rewards, and transitions into Python as mentioned above. This should make calculating a solution faster, but the option of dynamically converting rewards, states, and transitions from the wrapped (and therefore not duplicated) types as needed during calculation should be considered. I also did not provide a Python access to the VectorPolicy class, which would enable Policy Iteration for this assignment.

```
jl.seval("""
    generateForPy(m, s, a) = @gen(:sp, :r
                            )(m, s, a)
""")
sp, r = jl.generateForPy(self.m, convert(
                        self.state_type,
                        state), convert(self
                        .action_type, action
                        ))
```

Fig. 5: Calling the gen macro for HW3.

```
def evaluate(func, email = "", time=False
                            ):
    jl.seval("""
        HW3evaluatePy(fun::Py) = (m, s)
                                -> pyconvert
                                (Symbol,
                                pycall(fun,
                                m, s))
    """)
    jl.HW3.evaluate(jl.HW3evaluatePy(func
                            ), email, time=
                            time)
```

Fig. 6: Evaluating HW3's MCTS

D. Homework 3: Value Monte Carlo Tree Search

I was unable to complete a monte carlo tree search (MCTS) [6] solution in for homework 3, but I was able to prove that all the necessary functionality was present for a student to create a solution to this homework was (technically) possible. I was able to render an example dictionary through the browser tool, which successfully launched in Google Chrome. However, I had to get inventive with the gen macro for state transitions by creating an ad-hoc Julia function that replaced the macro (perhaps if I knew more about Julia I could avoid this, see figure 5). Similarly, the evaluate function required some creativity. I settled out of desperation rather than preference on creating another ad-hoc Julia function, that calls a python function, that then needs to convert a Julia MDP to a Python representation, that then calls a Julia function for state transitions while determining a solution as in figure 6. In short, a total and utter mess. For a solution to work credibly in Python for HW3 (especially if it is to be timed!), I believe at the very least a Python evaluate function should be created that passes a Python Mdp to the Python solution. Perhaps a full Python evaluation function for this problem is required.

E. Homework 4: Tabular Reinforcement Learning

As a example solution to HW4, I created a python implementation of SARSA [7]. This is the first of the CommonRlInterface.jl group, and as such the Julia environment representation was wrapped in a Python gymnasium interface. This opens up opportunities for students to explore python implementations of various RL algorithms written for this very popular package. As with the MDP group, the states and actions list are loaded into Python but operations such as act! and terminated are wrapped and called dynamically. My Python solution was able to match the provided Julia solution

almost exactly. However, once again there were performance issues. This could be an artifact of my solution (see above about numpy arrays as hashtable keys), but it could also be a function of once again relying on continuously going back to Julia for transition info. While loading the environment into Python took negligible time, training SARSA and then evaluating it as in the Julia HW4 took almost 15 minutes, where my Julia implementation took less than a minute with the same parameters.

F. Homework 5: Deep Reinforcement Learning

HW5 was unique in that a Python interface was provided with the homework description. I cleaned up the interface to make it like the rest of the homework packages, where all the Julia calls are handled through HW5.py. I also added functionality that renders the environment quickly enough that I am able to capture the state's evolution at every 20th trajectory while training. After attempting to implement Deep Q-Learning (DQN) [8] in TensorFlow, I eventually capitulated and found an example implemented in PyTorch [9]. PyTorch was able to easily find the requisite CUDA libraries to leverage the GPU (unlike TensorFlow) and as such training ran very quickly. Evaluation was another matter, taking about three minutes for a well-trained model vs a few seconds for a well-trained model in Julia. This model was able to produce a somewhat higher score than my Julia implementation.

Due to the conditional in DQN's loss function, it is non-trivial to design a performant solution in both PyTorch's and TensorFlow's (the two most common Python NN frameworks) explicitly tensor-heavy models. Julia's Flux, although slower (but without the advantage of CUDA in my case), also made it much easier to design a elegant DQN implementation along the lines of what was taught in class. While I think its valuable to provide a Python version of the assignment for students to apply other readily-available Python models to I find the learning experience implementing DQN from its basic theory to be much more effective in Flux.

G. Homework 6: POMDPs

Both Homework 6 and and Homework 5's POMDP section have so far been conspicuously absent from this report. Given the Julia POMDP framework's reliance on multiple dispatch and the performance degradation I found when working with Homeworks 3 and 4 I did not prioritize these sections. Finding a way to shoehorn Python functions into this framework would have been very interesting, perhaps more interesting than anything detailed up to this point. As such, I regret not giving it a serious effort. However, unless the performance degradation that comes from multiple calls back and forth from Julia to Python can be overcome, I do not believe that any amount of effort would have produced an acceptable solution. Perhaps flipping my paradigm around, calling Python functions from Julia instead of the other way around, would produce positive results. If the instruction staff is serious about a Python environment for this homework that is my recommendation for the first thing to try. If that avenue does not prove fruitful, the only other option I can recommend would be to provide an entire POMDP framework in native Python, though this case is obviously not attractive.

IV. Performance Evaluation

Throughout section III, a common theme was poor performance. After completing my implementations, it was still an open question as to whether or not this poor performance was due to inefficiencies in my Python code or inefficiencies in conversion between Julia and Python. Homework 4 provides an easy example of the performance difference, as part of the solution is to produce two graphs: one of environment steps vs. return and the other of wall clock time vs. return. These graphs can be seen in figure 7. The Python implementation, while taking an almost equal number of steps and performing almost as well, takes significantly longer to process.

This is not an immediate indictment of PythonCall/JuliaCall. It is entirely possible that the Python implementation, while accessing the data it needs in a perfectly Pythonic way, is still too Julian in structure as it was modeled after the Julia implementation. Therefore, I ran a series of trials on the environments in homeworks 3, 4, and 5. I left Homework 2 out from this trial as it was nearly as performant as the Julia version and also as after data is loaded into Python at initialization (the poor initialization performance is easily explained), the solvers for this environment do not interact much with Julia via function calls. For each environment for homeworks 3, 4, and 5 I performed a transition one million times and recorded the total time for all one million transitions in both Python and Julia. From this, we can see the average time a single transition will take and gain a notion of how much time a strenuous training or exploration sequence on each environment may take. For the grid world in HW3, this operation was generating a subsequent state from a random starting action. For the environments in homeworks 4 and 5, this was taking random actions until the environment terminated, then resetting and starting again until the one-million step count has been reached. The table for this can be seen in figure 8.

It is easy to see that, even when potentially inefficient Python implementations are removed from the calculation, the Python accesses to the environments are significantly more costly than those in the environments' native Julia.

V. Conclusion

It is easy to understand the position of the Python-proficient student at the beginning of this course who bemoans the fact that Julia must be used for the homework assignments. Its 1-based indexing, multiple dispatch, and various other idiosyncrasies may be at first off-putting to students that have so far in their academic journey heavily relied on Python or C-like languages for their classwork (perhaps outside of the odd programming languages class). However, after working through various problems in uncertainty, it is hard to ignore that Julia is very well suited to this kind of calculation. Its idiosyncrasies are what allow it

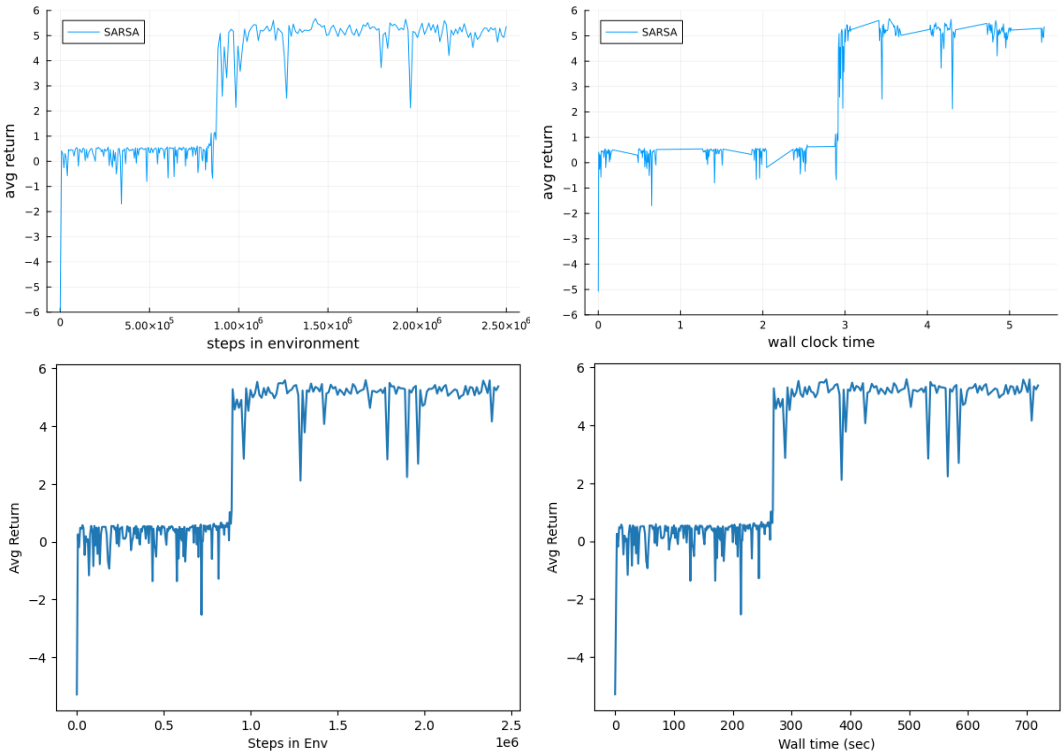

Fig. 7: The top row figures are the environment steps vs. average return and the wall clock time in seconds vs. average return for the Julia implementation of SARSA. The bottom row is the same figures for the Python implementation. Note that while the returns and number of steps in each implementation is consistent, the wall time is vastly different, the Julia implementation taking about 6 seconds to train where the Python version took over 700 seconds, or around twelve minutes.

Language	**HW3 total**	**HW3 avg**	**HW4 total**	**HW4 avg**	**HW5 total**	**HW5 avg**
Julia	65.266	7×10^{-5}	0.037	4×10^{-8}	0.069	7×10^{-8}
Python	627.36	6×10^{-4}	269.01	3×10^{-4}	18.75	2×10^{-5}

Fig. 8: Average and total times for one million transitions in each of the homework 3, 4, and 5 environments. Notice that the Python times are all significantly longer.

to be, after an initial learning curve, intuitive and performant (for example, efficiently iterating through a map keyed on a tuple by one item of that tuple). During the course of this project, I appreciated the handy code-completion, debugging, and assorted other tools that go hand-in-hand with a language as popular as Python. However, that did not make up for the ease that Julia in and of itself provides. This project is firmly in the proof-of-concept state and not ready for production, but even in its unpolished and occasionally inefficient state it did not instill any confidence that a Python environment for this course would be a benefit. Part of a computer science-adjacent education is learning how to work in diverse paradigms, so Julia's status as a less-familiar language should not be a detriment against it as an educational tool. Even setting aside the performance issues highlighted in figure 8, the difficulty evaluating Python solutions, and the main Python deep-learning frameworks' relative inflexibility, I see no reason to have a Python environment for Decision Making under Uncertainty.

VI. Future Work

If the course staff still wishes to continue pursuing a Python package for Decision Making Under Uncertainty, the first step would be to comb over DMUpy for things I may have missed in my inexperience that may make it more performant. The next step would be to polish it, adding dependency references and making it into something that does not need to be imported via relative paths.

As mentioned above, it could be that this package is attempting to solve the problem backwards by calling Julia from Python rather than Python from Julia. It would be worth attempting to reverse this. It would be harder for a student to debug their code, as in the case of a provided and theoretically more stable Julia environment calling a student's Python root causes may be harder to find. However, this may increase speed.

Finally, I am curious as to why the Homework 5 mountain car environment performed so much better for Python in the one million access test than the grid world for Homework 4. The mountain car environment with its length one-hundred state vectors should be more complex.

VII. Release

The author grants permission for this report to be posted publicly.

REFERENCES

[1] P. Krill. Julia language cracks top 20 in Tiobe popularity index. InfoWorld. [Online]. Available: https://www.infoworld.com/article/3704154/julia-language-cracks-top-20-in-tiobe-popularity-index.html

[2] "JuliaPy/PyCall.jl," JuliaPy. [Online]. Available: https://github.com/JuliaPy/PyCall.jl

[3] "JuliaPy/PythonCall.jl," JuliaPy. [Online]. Available: https://github.com/JuliaPy/PythonCall.jl

[4] Y. Aviv and A. Federgruen, "The value iteration method for countable state Markov decision processes," vol. 24, no. 5, pp. 223–234. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167637799000152

[5] I. P. Stanimirovic and M. B. Tasic, "Performance comparison of storage formats for sparse matrices," *FACTA UNIVERSITATIS (NI˘S) Ser. Math. Inform*, vol. 24, pp. 39–51, 2009.

[6] M. Aswiechowski, K. Godlewski, B. Sawicki, and J. Maadziuk, "Monte Carlo Tree Search: A review of recent modifications and applications," vol. 56, no. 3, pp. 2497–2562. [Online]. Available: https://doi.org/10.1007/s10462-022-10228-y

[7] G. A. Rummery and M. Niranjan, "On-line q-learning using connectionist systems," 1994. [Online]. Available: https://api.semanticscholar.org/CorpusID:59872172

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," vol. 518, no. 7540, pp. 529–533. [Online]. Available: https://www.nature.com/articles/nature14236

[9] A. Paszke and M. Towers. Reinforcement Learning (DQN) Tutorial PyTorch Tutorials 2.3.0+cu121 documentation. [Online]. Available: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html