PsychSim

Jun 20, 2022

Contents:

> "I never satisfy myself until I can make a mechanical model of a thing. If I can make a mechanical model, I can understand it. As long as I cannot make a mechanical model all the way through I cannot understand it."
>
> —Lord Kelvin

CHAPTER 1

Installing

1. Clone the repository:

```
git clone https://github.com/usc-psychsim/psychsim.git
```

2. Install system-wide by running the following from the installation directory:

```
pip install -e .
```

3. Profit

Uninstall:

```
pip uninstall psychsim
```

CHAPTER 2

Modeling

> "O brave new world,
> That has such people in't."
>
> – William Shakespeare (The Tempest, Act 5, Scene 1)

This is a world:

```
from psychsim.world import World
world = World()
```

A world has people:

```
from psychsim.agent import Agent
rufus = Agent('Rufus')
world.addAgent(rufus)
```

And groups of people:

```
free = Agent('Freedonia')
world.addAgent(free)
```

And killer robots:

```
world.addAgent(Agent('Gort'))
```

These are *agents*, because they have names:

```
for name in world.agents:
   agent = world.agents[name]
   print(agent.name)
```

Of course, not everything that has a name is an agent. There are a number of reasons to make something an agent. The most common one is that something *is* an agent, i.e., an autonomous, decision-making entity. But there can be good reasons for modeling non-agents as agents within PsychSim, usually because of improved readability of simulation output. Conversely, not every agent has to be modeled as an agent, especially if that agent's decision-making is not

relevant to the target scenario. It does not necessarily cost anything to have extra agents in the scenario (if they do not get a *turn*, their decision-making procedure is never invoked). The choice of whether to make something an agent is yours.

2.1 State

The *state* of a simulation captures the dynamic process by which it evolves. As in a *factored MDP*, we divide the state, $\vec{S}$, into a set of orthogonal features, $S_0 \times S_1 \times \cdots \times S_n$. Each feature captures a distinct aspect of the simulation state.

2.1.1 Unary State Features

It can be useful to describe a state feature as being local to an individual agent. Doing so does *not* limit the dependency or influence of the state feature in any way. However, it can be useful to define state features as local to an agent to make the system output more readable. For example, the following defines a state feature "troops" for the previously defined agent "Freedonia":

```
troops = world.defineState(free.name, 'troops')
```

The return value of `defineState()` is the unique symbol that PsychSim assigns to this newly defined state feature. To set the value for this state feature, you can either use this symbol with `setFeature()`, the original agent/feature combination with `setState()`, or the feature name with `setState()`. In other words, the following three statements are completely interchangeable:

```
world.setFeature(troops, 40000)
world.setState(free.name, 'troops', 40000)
free.setState('troops', 40000)
```

For state features which are not local to any agent, a special agent name (i.e., `WORLD`) indicates that the state feature will pertain to the world as a whole. Again, from the system's perspective, this makes no difference, but it can be useful to distinguish global and local state in presentation:

```
treaty = world.defineState(WORLD, 'treaty')
```

Thus, one reason for creating an agent is to group a set of such state features under a common name, as opposed to leaving them as part of the global world state.

2.1.2 Binary State Features

There can also be state features that represent the *relationship* between two agents:

```
freeTrustsSyl = world.defineRelation(free.name, 'Sylvania', 'trusts')
```

The order in which the agents appear in this definition *does* matter, as reversing the order will generate a reference to a different element of the state vector. For example, if the previous definition corresponds to how much trust Freedonia places in Sylvania, the following variation would correspond to how much trust Sylvania places in Freedonia:

```
sylTrustsFree = world.defineRelation('Sylvania', free.name, 'trusts')
```

The values associated with these relationships can be read and written in the same way as for unary state features. However, there are no helper methods like `setState()` or `getState()`. Rather, you should use the symbol returned by `defineRelation()` in combination with `setFeature()`:

```
world.setFeature(freeTrustsSyl, 0.25)
world.setFeature(sylTrustsFree, 0.75)
```

2.1.3 Types of State Features

By default, a state feature is assumed to be real-valued, in $[-1, 1]$. However, these state features are one example of PsychSim's more general class of random variables. These variables support a variety of domains:

float real valued and continuous

int integer valued and discrete

bool a binary `True`/`False` value

list/set an enumerated set of possible values (typically strings)

By default, a variable is assumed to be float-valued, so the previous sections definitions of state features created only float-valued variables. Both the `defineState()` and `defineRelation()` methods take optional arguments to modify the domain of valid values of the feature. The following definition has the identical effect as the previous trust definition, but it makes the default values for the variable type and range of possible values explicit:

```
freeTrustsSyl = world.defineRelation(free.name, 'Sylvania', 'trusts', float, -1, -1)
```

This relationship can now distinguish between a trusting and distrusting relationship (positive vs. negative values), with a fine-grained magnitude of the degree of (dis)trust. It is also possible to specify that a state feature has an integer-valued domain instead:

```
troops = world.defineState(free.name, 'troops', int, 0, 50000)
```

One can also define a boolean state feature, where no range of values is necessary:

```
treaty = world.defineState(WORLD, 'treaty', bool)
```

It is also possible to define an enumerated list of possible state features. Like all feature values, PsychSim represents these numerically within the actual state, but you do not need to ever use the numeric values:

```
phase = world.defineState(WORLD, 'phase', list, ['offer', 'respond', 'rejection', 'end
↪', 'paused', 'engagement'])
world.setState(WORLD, 'phase', 'rejection')
```

2.2 Actions

The most common reason for creating an agent is to represent an entity that can take *actions* that change the state of the world. If an entity has a deterministic effect on the world, you can define a single action for it. However, agents typically have multiple actions to choose from, and it is the decision among them that is the focus of the simulation.

2.2.1 Atomic Actions

The *verb* of an individual action is a required field when defining the action:

```
reject = free.addAction({'verb': 'reject'})
```

The action created will also have a *subject* field, representing the agent who is performing this action. The *subject* field is automatically filled in with the name of the agent ("Freedonia" in the above example). A third optional field, *object*, can represent the target of the specific action:

```
battle = free.addAction({'verb': 'attack', 'object': 'Sylvania'})
```

An action's field values can be accessed in the same way as entries in a dictionary:

```
if action['verb'] == 'reject' and action['object'] == 'Sylvania':
   print(f'Sylvania has been rejected by {action["subject"]}')
```

You are free to define any other fields as well to contain other parameterizations of the actions:

```
offer50 = free.addAction({'verb': 'offer','object': sylv.name, 'amount': 50})
```

We will describe the use of these fields in *Dynamics*.

2.2.2 Action Sets

Sometimes an agent can make a decision that simultaneously combines multiple actions into a single choice:

```
rejectAndAttack = free.addAction([{'verb': 'attack','object': sylv.name},
                                  {'verb': 'reject'}])
```

For the purposes of the agent's decision problem, this option is equivalent to a single atomic action (e.g., simultaneous rejection and attack). However, as we will see in *Dynamics*, separate atomic actions can sometimes simplify the definition of the effects of such a combined action.

The return value of `addAction()` is an `ActionSet`, even if only one atomic `Action` is specified. All of an agent's available actions are stored as a set of `ActionSet` instances within an agent's `actions`. An `ActionSet` is a subclass of *set*, so all standard Python set operations apply:

```
for action in free.actions:
   print(len(action))
rejectAndAttack = reject | battle
```

By default, an agent can choose from all of its available actions on every turn. However, we may sometimes want to restrict the available action choices based on the current state of the world. We will cover how to specify such restrictions in *Legality*. As a result, rather than inspecting the `actions` attribute itself, we typically examine the context-specific set of action choices instead:

```
for action in free.getActions():
   if len(action) == 1:
      print(action['verb'])
```

The fragment above illustrates one helpful shortcut for `ActionSet` instances: you can access fields within the member actions as long as all of the member actions have the same value for that field. In other words, `rejectAndAttack['subject']` would return `'Freedonia'`, but `rejectAndAttack['verb']` would raise an exception.

2.3 Probability

Maybe you already know this, but uncertainty is everywhere in social interaction. As a result, `Distribution` objects are central to PsychSim's representations. Probability distributions can be treated as dictionaries, where the

keys are the elements of the sample space, and the values are the probabilities associated with them. For example, we can represent a fair coin with the following distribution:

```
coin = Distribution({'heads': 0.5, 'tails': 0.5})
if coin.sample() == 'heads':
   print('You win!')
```

If you happen to lose enough that you suspect that the coin is in fact *not* fair, then you can update your beliefs by changing the distribution:

```
coin['heads'] = 0.25
coin['tails'] = 0.75
```

If you want to know the probability that the coin lands on its edge, `coin['edge']` would throw an exception, while `coin.get('edge')` would return 0. To account for the nonzero probability that the coin lands on its edge, you must explicitly add such a probability:

```
coin['edge'] = 1e-8
coin.normalize()
for element in coin.domain():
   print(coin[element])
```

2.4 Piecewise Linear (PWL) Functions

The `Distribution` is sufficiently expressive for our needs, but it useful to impose additional structure on the sample space to facilitate authoring, simulation, and understanding. As already mentioned, PsychSim uses a factored representation, so that a state of the world is expressed a probability distribution over possible feature-value pairs. More precisely, instead of distributions over arbitrary elements, PsychSim represents distributions over `Keyedvector` instances, representing a set of feature-value pairs. The features are the same unique identifiers created by functions like `stateKey()` and returned by methods like `defineState()`:

In particular, the state of the world is represented as a `VectorDistributionSet` that represents a probability distribution over possible worlds:

```
world.setState(WORLD, 'phase', 'engagement')
world.setState(WORLD, 'winner', Distribution({'Sylvania': 0.25, 'Freedonia': 0.75}))
free.setState('troops', Distribution({10000: 0.25, 25000: 0.75}))
```

These statements declare that the simulation is currently in the *engagement* phase, with a 75% chance that the winner is Freedonia vs. a 25% chance that it is Sylvania, and with Freedonia having a 75% chance of having 25000 troops vs. a 25% chance of having 10000. These three state features have independent distributions within the state. In this state, the probability that Freedonia is the winner with 25000 troops remaining is 56.25%.

If we want to instead specify that Freedonia has 25000 troops if and only if it is the winner, then we specify a joint probability over *winner* and *troops*. To do so, we use a `KeyedVector` to represent elements of the joint sample space:

```
freeVictory = KeyedVector({stateKey(WORLD, 'winner'): 'Freedonia',
                           stateKey(free.name, 'troops'):): 25000})
sylvVictory = KeyedVector({stateKey(WORLD, 'winner'): 'Sylvania',
                           stateKey(free.name, 'troops'):): 10000})
```

over possible worlds. Thus, even though the above method call specificies a single value, the value is internally represented as a distribution with a single element (i.e., 40000) having 100% probability. We can also pass in a distribution of possible values for a state feature:

The `Distribution` constructor takes a dictionary whose keys constitute the distribution's sample space, and whose values constitte the probability mass of each element of that space. In the above example, the distribution over Freedonia's cost is 50-50 between 1000 and 2000. When you call the `setState()` method with a probabilistic value, PsychSim *joins* the new distribution with the current state vector. After the previous two `setState()` calls, there will be two possible worlds, each with 50% probability: one where Freedonia has 40000 troops and cost 1000, and a second where Freedonia has 40000 troops and cost 2000. Just as the second call doubles the number of possible worlds, a subsequent call to `setState()` with a probabilistic value will similarly increase the number of possible worlds by a factor equal to the size of the distribution passed in. In other words, calling `setState()` will generate worlds for all possible combinations of the individual values for the state features.

If you would like more fine-grained control over the possible worlds, simply manipulate the distribution directly. Note that the world state is potentially a dictionary of distributions over worlds, although until further development occurs, the only entry in that table is indexed by {tt None}::

```
possworld1 = KeyedVector({stateKey(free.name, 'troops'): 40000,
                          stateKey(free.name, 'cost'): 1000})
possworld2 = KeyedVector(possworld1)
possworld2[stateKey(free.name, 'cost')] = 2000
possworld3 = KeyedVector()
possworld3[stateKey(free.name, 'troops')] = 25000
possworld3[stateKey(free.name, 'cost')] = 2000

world.state[None].clear()
world.state[None][possworld1] = 0.1
world.state[None][possworld2] = 0.4
world.state[None][possworld3] = 0.5
```

When querying for a given state feature, the returned value is *always* in `Distribution` form.:

```
value = world.getState(free.name,'phase')
for phase in value.domain():
   print('P(%s=%s) = %5.3f' % (stateKey(free.name,'phase'),
                               phase,value[phase]))
```

The `stateKey()` function is useful for translating an agent (or the world) and state feature into a canonical string representation:

```
from psychsim.pwl import *
s = KeyedVector({'S_0': 0.3, 'S_1': 0.7})
s['S_n'] = 0.4
for key in s:
   print(key, s[key])
```

Notice that PsychSim allows you to refer to each feature by a meaningful *key*, as in Python's dictionary keys. Keys are treated internally as unstructured strings, but you may find it useful to make use of the the following types of structured keys.

PsychSim uses piecewise linear (PWL) functions to structure the dependencies among variables, as we will see in later sections. While the PWL structure limits the expressivity of these dependencies, it provides a more human-readable language (as opposed to arbitrary code) and, more importantly, provides invertibility that is essential for automatic fitting and explanation.

We have already seen the basic building block of the PWL functions, the {tt KeyedVector}.

2.4.1 Legality

Legality:

```
tree = makeTree({'if': equalRow(stateKey(WORLD, 'phase'), 'offer'),
                 True: True,
                 False: False})
free.setLegal(action, tree)
```

2.4.2 Dynamics

2.4.3 Termination

Termination conditions specify when scenario execution should reach an absorbing end state (e.g., when a final goal is reached, when time has expired). A termination condition is a PWL function (Section ref{sec:pwl}) with boolean leaves.:

```
world.addTermination(makeTree({'if': trueRow(stateKey(WORLD, 'treaty')),
                               True: True, False: False}))
```

This condition specifies that the simulation ends if a "treaty" is reached. Multiple conditions can be specified, with termination occurring if any condition is true.

2.5 Reward

An agent's *reward* function represents its (dis)incentives for choosing certain actions. In other agent frameworks, this same component might be referred to as the agent's *utility* or *goals*. It is often convenient to separate different aspects of the agent's reward function:

```
goalFTroops = maximizeFeature(stateKey(free.name, 'troops'))
free.setReward(goalFTroops, 1)
goalFTerritory = maximizeFeature(stateKey(free.name, 'territory'))
free.setReward(goalFTerritory, 1)
```

2.6 Models

A *model* in the PsychSim context is a potential configuration of an agent that may apply in certain worlds or decision-making contexts. All agents have a "True" model that represents their real configuration, which forms the basis of all of their decisions during execution.

It also possible to specify alternate models that represent perturbations of this true model, either to represent the dynamics of the agent's configuration or to represent the perceptions other agents have of it:

```
free.addModel('friend')
```

2.6.1 Model Attribute: *static*

2.7 Observations

CHAPTER 3

Simulating

CHAPTER 4

Indices and tables

- genindex
- modindex
- search