The Young Modeler's Handbook

David V. Pynadath

December 19, 2014

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.

LORD KELVIN

1 Worlds

O brave new world,
That has such people in't.

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('Klaatu'))
```

These are *agents*, because they have names.

```
for name in world.agents.keys():
   print world.agents[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.

1.1 State

The *state* of a simulation captures the dynamic process by which it evolves. As in a *factored MDP*, we divide an overall state vector, $\vec{S}$, 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.

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

1.1.1 Unary Keys

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".

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

For state features which are not local to any agent, a null agent name (i.e., `None`) 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.

```
world.defineState(None,'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. You can then specify the value for a given agent's corresponding state feature.

```
free.setState('troops',40000)
```

The world state is actually a `Distribution` 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.

```
free.setState('cost',Distribution({1000: 0.5, 2000: 0.5}))
```

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 `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.

1.1.2 Binary Keys

There can also be state that pertains to the *relationship* between two agents.

```
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.

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

The values associated with these relationships can be read and written in much the same way as for unary state features. However, there are no helper methods like “setState” and “getState”. Rather, you should use the direct method for manipulating state feature values.

```
key = world.defineRelation(free.name,'Sylvania','trusts')
world.setFeature(key,0.)

key = binaryKey(free.name,'Sylvania','trusts')
distribution = world.getFeature(key)
for value in distribution.domain():
   print 'P(%s=%5.3f) = %5.3f' % (key,value,distribution[value])
```

1.2 Variables

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)

ActionSet: enumerated set of actions

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 valid ranges of the feature. The range of possible values does not affect the execution of the simulation, as the simulation normalizes the values back to the $[-1, 1]$ range during decision-making.

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

It is also possible to specify that a state feature has an integer-valued domain with the optional “domain” argument.

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

One can also define a boolean state feature using the same argument, in which case the optional “lo” and “hi” arguments are obviously moot.

```
world.defineState(None,'treaty',bool)
```

It is also possible to define an enumerated list of possible state features. Like all feature values, PsychSim represents these as floats internally, but it automatically handles the translation back and forth.

```
phases = ['offer','respond','rejection','end','paused',
          'engagement']
world.defineState(None,'phase',list,phases)

world.setState(None,'phase','rejection')

distribution = world.getFeature('phase')
for phase in distribution.domain():
   print 'The phase is %s with probability %5.3f' % \
         (phase,distribution[phase])
```

The domains also work within "defineRelation" calls as well. In other words, relationships can take on the same sets of values as unary state features. Both "defineState" and "defineRelation" use the more general "defineVariable" method to set up the domains of their corresponding variables. This method can also be used directly to define a variable (but *without* creating a corresponding column in the state vector).

```
world.defineVariable(stateKey(free.name,'troops'),
                     domain=int,lo=0,hi=50000)
```

2 Actions

The most common reason for creating an agent is because there is 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.

2.1 Atomic Actions

The `verb` of an individual action is a required field when defining the action.

```
freeReject = 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.

```
freeBattle = free.addAction({'verb': 'attack','object': sylv.name})
```

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

```
if action['verb'] == 'reject' and action['object'] == sylv.name:
   # Sylvania will feel rejected
```

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

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

We will describe the use of these fields in Section 3.2.

2.2 Action Sets

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

```
free.addAction(set([{'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 Section 3.2, separate atomic actions can sometimes simplify the definition of the effects of such a combined action.

When inspecting an agent's action choice, each option is an `ActionSet` instance, supporting all standard Python `set` operations.

```
for action in free.actions:
   print len(action)
freeRejectAndAttack = freeReject | freeBattle
```

As we will see in Section 3.1, not all of the agent's choices may be legal under all circumstances. Rather than inspecting the `actions` attribute itself, we typically examine the context-specific set of action choices instead.

```
for vector in world.state[None].domain():
   print free.getActions(vector)
```

3 Piecewise Linear (PWL) Functions

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 `KeyedVector`.

3.1 Legality

```
tree = makeTree({'if': equalRow(stateKey(None,'phase'),'offer'),
                 True: True,
                 False: False})
free.setLegal(action,tree)
```

3.2 Dynamics

3.3 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 7) with boolean leaves.

```
world.addTermination(makeTree({'if': trueRow(stateKey(None,'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.

4 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.)
```

5 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')
```

5.1 Model Attribute: `static`

6 Observations

7 PWL functions

8 Executing a Scenario

The basic simulation method is the `step`, which generates a single decision epoch's worth of action choices, state changes, and belief updates. Calling the method with no arguments will allow all of the agents to choose their actions autonomously based on their beliefs in the current world.

```
world.step()
```

9 Algorithms

9.1 Decision Making

9.2 Belief Update

World has $\Pr(M_0)$

World generates an observation ω

But M_0 cannot generate ω

$$\begin{aligned}
\Pr(M_1 = m) &= \Pr(M_1 = m|M_0, \omega) && (1)\\
&= \frac{\Pr(M_1 = m, \omega|M_0)}{\Pr(\omega|M_0)} && (2)\\
&\propto && (3)
\end{aligned}$$

And done.