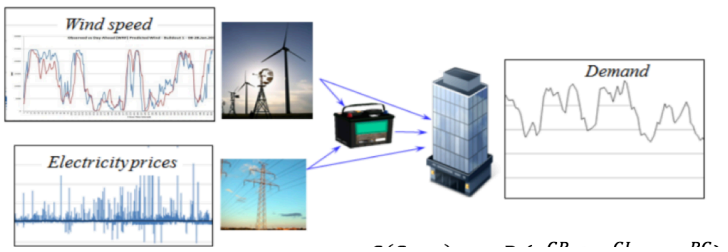
A Guide to Operation of the SMART Storage Policy Benchmarking Package

Author: Joe Durante

1. *Overview*

This package provides a customizable testing environment for energy storage control policies. A quick reference for the type of problems this package is designed to handle, one should read the paper named 'Powell-TutorialPartII-StochasticOptApril112015' located in the 'docs' folder of the package. Through the use of a spreadsheet, one may use one of the pre-programmed models for variables present in a basic energy storage problem (such as renewable sources, storage devices, loads, and electricity prices) to define an energy storage problem. Or, if desired, a template is available for design of a custom model. Custom models and pre-programmed models can be used for different variables together in the same problem. After defining other aspects of the problem, such as reward functions, a discount factor, and constraints, the performance of different policies for control of the system can be compared. Policies can be designed according to a template. There is also a benchmarking option available which first discretizes the problem and then finds an optimal policy based on solving the full Markov Decision Process (MDP). In addition, both backward and forward Approximate Dynamic Programming algorithms (used to find value functions for system states) are available to develop policies as well.

Throughout the manual we will be using the following problem as an example for how one would use the package:



- $\max_{\pi \in \Pi} \mathbb{E}^{\pi}\left[\sum_{t=0}^{T} C\left(S_t, X^{\pi}(S_t)\right) | S_0\right]$ where
- $C(S_t, x_t) = -P_t(x_t^{GR} + x_t^{GL} - \eta x_t^{RG})$ and
- $S_t^x = S^{M,x}\left(S_t, X_t^{\pi}(S_t)\right)$
- $S_{t+1} = S^{M,W}(S_t^x, W_{t+1})$

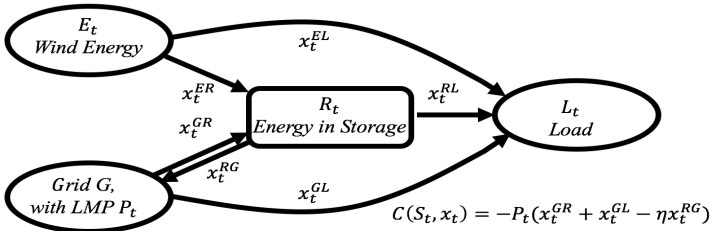


Figure 1. Example Energy Storage Problem

2. *The User Input File and main script*

In the 'input' folder of the package, there is a blank template for the user input file named 'User_Input.csv' along with example files 'Example_User_Input_Pol_Compare.csv' and 'Example_User_Input_Full_MDP.csv'. We will be using these examples to show how to operate the package. Additionally, in the 'src' folder, there is a script named 'main.py' which reads the input files and runs the different modules contained in the package. The first step a user should follow is to go into the 'main.py' script and set the variable InputFile, found in the first lines following the import statements (approx. lines 100-140), to the file path of the user input file to be read, shown below:

```
 95
 96 import numpy as np
 97 import csv
 98 import ast
 99 import os
100 #import random
101
102 #random.seed(7)
103
104 def try_parse(string):
105     try:
106         return float(string)
107     except Exception:
108         return np.nan
109
110 def try_parse_int(string):
111     try:
112         return int(string)
113     except Exception:
114         return np.nan
115
116 os.chdir("..")
117 os.chdir("input")
118
119 #USER INPUT
120 InputFile="Example_User_Input_Pol_Compare.csv"
121
```

Figure 2. Location of InputFile variable in Main.py

By default it is set to the path of the 'Example_User_Input_Pol_Compare.csv' file. The input files have 6 headers in the top row which should not be altered: 'Identifier', 'Type', 'Name', 'FilePath', 'Headers', and 'Parameters'. Defining each aspect of the problem (Nodes, Decisions, Constraints, etc.) entails populating an empty row beneath the headers with values. Depending on the aspect, not all the fields must be filled, and the aspect identifier placed under 'Identifier' will determine the meaning (and available options) of what is placed in each field. The 'Example_User_Input_Pol_Compare_2.csv' file is shown below:

	Identifier	Type	Name	FilePath	Headers	Parameters
2	Horizon					4
3	Discount Factor					1
4	Node	Exogenous Crossing State HSMM	E	Test_Wind_4.csv	dummy; Actual; Fcst	10;1;4000; 3; 3
5	Node	Exogenous Crossing State HSMM Prices	G	TestElectricityPrices.csv	dummy; Price; dummy; Temp	1;1;'ObsMax';[0,0.75,1.0];[0,0.3,0.8,1.0];4;5
6	Node	Basic Resource Node	R_1			2000;10000; 2000;1; 1
7	Node	Exogenous Deterministic Node	L	TestElectricityPrices.csv	dummy; dummy; Data; dummy	1;1;5000
8	New Forecast		E	Test_Wind_4.csv	dummy; dummy; Fcst	
9	decision		G:R_1			10
10	decision		G:L			10
11	decision		R_1:L			10
12	decision		E:R_1			10
13	decision		E:L			10
14	Action Space	Example Decision Space				
15	Constraints_and_Rewards	Example Constraints and Rewards				
16	Policy	PolA	PFA_1			15;18
17	Policy	PolA	PFA_2			15;20
18	Policy	VFA_Post_Parametric	VFA_PostParAPI	fADP_Lin_VFA_4		
19	Policy	VFA_Pre_Parametric	VFA_PrePar	bADP_Lin_VFA		
20	Policy	VFA_Post_Lookup	VFA_LU	bADP_LU_Table		
21	Module	Policy Compare		Pol_Compare_2		5;['E','R_1','L','G'];[1,0,0,1];['G:R_1','G:L','R_1:L','E:R_1','E:L'];[0,0,0,0,0]

Figure 3. Example_User_Input_Pol_Compare_2.csv

The following sections describe what options are available to be placed in each of these fields to use the user input file, the 'main.py' script, and, if desired, custom scripts to operate the package. To use the package, the final step is to run the 'main.py' script.

3. Identifier Field

The following text options may be placed in the identifier field of the user input file to describe the aspect of the problem we are describing: Horizon, Discount Factor, Node, New Forecast, Decision, Constraints_and_Rewards, Action Space, Policy, or Module.

4. Filling in a row with the 'Horizon' Identifier

This is the optimization horizon for the problem, in terms of the number of time steps in the program. Set 'Params' field to an integer that will set the optimization horizon. between 0 and 1.

5. Filling in a row with the 'Discount Factor' Identifier

If one wants to discount future rewards, create a row for the discount factor and set the 'Params' field to a factor between 0 and 1. Otherwise, the default is 1.

6. Filling in a row with the 'Node' Identifier

One has the option to use a built-in node type (with possibly different parameters) or create a custom node. Custom nodes must adhere to a template and several required functions must be defined. These will be described later. First, we explain the built-in nodes, which model common node types in an energy storage problem/network. There are 4 types of built-in nodes: a node for a basic storage device (ex. a battery), a node for an exogenous process with a forecast which will form a unique stochastic hidden semi-Markov model (HSMM) which replicates crossing times, a node for an exogenous price process which will form a unique stochastic hidden semi-Markov model conditioned on a temperature forecast tailored to produce simulations of electricity prices, and a deterministic model where the exogenous variable takes on values according to an input file. The HSMMs are described in depth in the pdf titled 'CrossingTimeMarkovModels' and 'Durante_Energy_Storage_Backward_ADP_10_10_17' located in the 'docs' folder of the package.

First, the 'Name' field is crucial to the operation of the package. It carries a user defined name that will identify the Node throughout the package. There only restrictions are that no node may be named 'T' (used to keep track of the time in the program) and no two nodes should have the same name. Each node must be given a name.

In the 'Type' field, enter 'Basic Resource Node' for the basic storage device node. The remaining fields can be left blank, except for the 'Params' field. This should contain 5 entries, separated by semicolons. The first is the discretization interval for the states which the battery may take on (number > 0). The discretization interval should be set low for nearly continuous, and high (relative to maximum value of the process) for highly discretized; larger values will result in shorter MDP (and ADP) runtimes. The second is the is the maximum charge/discharge rate of the battery per time period, the third is the round trip energy conversion efficiency (number between 0 and 1, represents fraction of energy transmitted/received to/from other nodes in the network that is available for use/charge), the fourth is the leakage/loss rate per time period (number between 0 and 1, represents fraction of charge lost per time period). Note in this model, conversion rate, leakage, and charge rates are constant, though one

might want to change the model such that these are stochastic processes. Similarly, in the case of a hydroelectric dam for instance, the amount of available energy may undergo a stochastic change (from rainfall/evaporation) separate from the decision to draw water out/pump water in. In this case, one should create a custom node or alter the existing 'basic_resource_node.py' script to accommodate this.

In the 'Type' field, enter 'Exogenous Crossing State HSMM' for the standard Crossing State HSMM. This requires data to be read into the program to fit the model, enter the file path of a .csv file containing data in the 'FilePath' field. The data must contain a column with a forecast and a column with the actual realized data, with or without headers. It is acceptable to have more columns with different data present (such as a time stamp) as well. In order to properly read in the data, the 'Headers' field should have n arguments separated by semicolons where n is equal to the number of columns in the data file. The position of the column with the forecast data must be identified with the label 'Fcst', while the position of the column with the actual power produced must be identified with 'Actual'. The remained can be identified with any string; the examples use 'dummy'. An example of this entry procedure is shown in the Figure 3, where we read in data from the file 'Test_Wind_4.csv'. In the data file, there are three columns, the first being a time stamp on the data, the second being the realized wind power data and the third being the forecasted wind power for that time. Thus our 'Headers' field reads: 'dummy; Actual; Fcst'. The 'Params' field should contain 5 numbers separated by semicolons. The first is the discretization interval for the values which the process may take on (number > 0). The discretization interval should be set low for nearly continuous, and high (relative to maximum value of the process) for highly discretized; larger values will result in shorter MDP (and ADP) runtimes. The second parameter should be an integer representing how many time steps it takes the process to transition relative to the fastest transitioning process. The quickest transitioning process should thus have the integer 1 for this parameter (If all nodes transition at each time step, then set this =1). The third parameter is a scaling factor representing the maximum value of this node. The fourth is integer representing the number of run length quantiles in the model (refer to pdf for explanation) and the fifth is an integer representing the number of aggregated states for forecast errors (refer to pdf). For the last two entries, if unsure of values to use, one can leave these blank as they are preset if not defined.

In the 'Type' field, enter 'Exogenous Crossing State HSMM Prices' for the Crossing State HSMM designed specifically for electricity price modelling. Again, enter the file path for the data to be read in. In this case, the model requires actual data and a temperature forecast (or temperature data) arranged in columns in a .csv file. The entry in the 'Headers' field is as previously described, except 'Price' identifies the position of the price column, while 'Temp' identifies the position of the temperature column. The 'Params' field should contain 7 entries separated by semicolons. The first 3 entries are as in the 'Exogenous Crossing State HSMM' model, where the third entry is now the maximum price (enter 'ObsMax' to use the observed maximum), and scales all the data relative to this. The fourth is a list in the form of [0, 0.xx, … ,0.xx, 1], where 0.xx represents a decimal between 0 and 1. This determines the division points for the temperature seasonality bins (relative to maximum) which are used in the model for conditioning price distributions (refer to pdf for description). The fifth is a list formed in the same manner, but for the division of temperature trend bins (refer to pdf). The sixth entry is an integer that determines the number of aggregated price states (refer to pdf). The seventh entry is the number of minutes per time step; this is important as daily trends are observed in electricity prices and we leverage that in the model. The fifth entry is the discretization interval and the sixth entry is the time step. For the final four entries, if unsure of values to use, one may omit these as they are present, however it should be noted the presets are tuned for summer electricity price data (Locational Marginal Price Data

in 5 minute increments) in Princeton, NJ, USA, and for other locations or periods during the year, may need to be tuned to produce reflect the change in price dependence on temperature (for example, during the winter, prices may spike when the temperature trend is colder, not hotter, and thus an adjustment should be made).

In the 'Type' field, enter 'Exogenous Deterministic Node' for the deterministic model. This model is used for exogenous variables, that, to simplify the problem, are to be deterministic and follow a single pre-determined path over time. The 'FilePath' field is populated as before. Only one column of data is to be read in, defining the values for which the process will take on at each time step. The 'Headers' field is populated as described earlier as well, with 'Data' identifying the position of the relevant data column. In the 'Params' field, enter the discretization interval, though this can be very small as it will not have a major effect on any runtimes (for any module currently implemented in the package), a semicolon, the time step, another semicolon, and then the maximum value of the node (the data will be scaled relative to this maximum).

If a custom model is desired, enter 'Custom Node x' where x is either 1,2 or 3, and create a custom model according to the template provided. Note if a custom model is chosen, extra steps must be taken to use the model. The template 'Custom_Node_x.py' should then be filled in with all functions defined. See http://adp.princeton.edu/Papers/Powell_ADP_2ndEdition_Chapter%205.pdf for information about pre- and post- decision states as well as transition functions and state variables. The remaining fields may be used to pass information to the node as one sees fit. A custom process can be used more than once with different names and parameters, however if more than three types of custom exogenous process models are desired, additional steps are needed to import and read-in the process. However, if the pattern in the 'main.py' file is followed, this is relatively simple. First the script with the process must be imported at the top of 'main.py'. Then, an additional conditional statement must be added to 'main.py' under the Node read-in section, specifying that the new 'Custom_Node_y.py' (y>3) should be read in and used. This procedure is the same for all custom types throughout the remainder of the package.

7. Filling in a row with the 'New Forecast' Identifier

This is a fairly straightforward row to fill in. One would use this if they want to train a model based on data from history and then use the model given new forecasts for the process. The 'Name' field should contain the name of the node for which new forecasts are available. The FilePath field should have the file path of the file with the new forecast. The Headers field should be filled in as before, except only the column containing the forecast should be labelled 'Fcst', while the rest have the word 'dummy' as place holders.

8. Filling in a row with the 'Decision' Identifier

This is the way to identify the links in the network, energy flow paths, or in general, any possible decisions. The 'Name' field needs to be populated; one decision variable per row. For an energy flow variable from one node to another, use the format 'from_node_name:to_node_name' where the names originate from the variables defined with the Node Identifiers. For example, 'G_1:R_1' or 'R:E_wind' would be acceptable flows given each variable was defined somewhere in the input file. Additionally the 'Params' can be filled in with a positive integer representing the maximum number of (approximately) evenly spaced actions for the flow variable considered when finding feasible actions. This is only used if an 'Action Space' which utilizes this information is selected (see section 10 for explanation), however a

number must be placed to avoid an error. General decision can be named as one sees fit as long as they are compatible with the rest of the aspects of the problem (ex. they are defined the same way in the reward function).

9. Filling in a row with the 'Constraints_and_Rewards' Identifier

Reward functions and constraints are problem specific and thus there are no built-in forms of constraints and rewards, there is an example script, 'Example Constraints and Rewards' however. However, there are specific guidelines to what constraints and reward functions must produce in the program. Entering constraints and reward functions will require some Python knowledge and one must fill in a template Python script, but this step cannot be completed effectively through solely the use of a spreadsheet. Set the 'Type' field such that the correct constraints and rewards file is read in for the problem (in Figure 3, we set it to 'Example Constraints and Rewards') and follow the instructions outlined in the template 'custom_constraint_and_reward_entry.py' to fill in each required function (also outlined below).

Constraint Entry Instructions:

- First return a list of all decision/flow variables that are involved in an equality constraint in a list under the equality_constraint_variables function, using the decision variable names previously defined. Create a function which then accepts, in order: state,decision,nodes, and return a dictionary in which the slack variable(s) are returned with the decision variable name as the key and the possible value as the dictionary value. Return this function from equality_constraints. Usually in these type of energy storage problems, the 'G:L' flow variable is the slack variable that ensures the load demand is met. Thus for this example, we return {'G:L': possible_value}. Here, decisions is a dictionary containing decision/flow variables as keys, and a possible value for each as the dictionary values. To access the possible value inside the function, type decisions['decision_name']. The nodes entry is a dictionary in which the node variables are keys and the nodes themselves are values. To access a node value, use nodes['node_name'].get_preds_value(state). Available also are all the functions present in each node.

- Under remaining constraints, create functions for any inequality constraint which accepts, again, state, decisions,nodes (the same variables as the Equality_Constraints function) but only return a Boolean value for each function: True if the decisions/current state combo satisfies the constraint and False otherwise. Once the inequality constraint functions are all formed, place the functions into a Python list which is returned in the function remaining_constraints

Reward Function Entry Instructions:

- Create a function that accepts as arguments, in order, state, decisions, nodes. The description of each is as above. The function must then return a single number for the reward realized by the state, decision pair. Note the package is designed to maximize

cumulative rewards, not minimize cost, so the reward function should be designed with this in mind. Return this function under from the function reward_function.

The example file 'Example_Constaint_and_Reward_Entry.py' is shown below:

```python
from Constraint_and_Reward_Function_Entry import Constraint_and_Reward_Function_Entry

class Example_Constraint_and_Reward_Entry(Constraint_and_Reward_Function_Entry):

    def equality_constraints(self):
        """
        Return a single function for finding slack variable(s) in each equality constraint
        (typically of the G:L type) that returns a dicitonary as follows:

        Inputs to functions are, in order: decision, energy_vars, resource_vars, state

        {slack_decision_variable_1:equality constraint solving for it,
        slack_decision_variable_2:equality constraint solving for it}

        """

        def Equality_Constraints(state,decision,nodes):
            return {'G:L':nodes['L'].get_preds_value(state)-decision['E:L']-(nodes['R_1'].conv_loss*decision['R_1:L'])}

        return Equality_Constraints

    def equality_constraint_variables(self):
        """
        Return list of all decision variables involved in an equality constraint
        """
        return ['E:L','G:L','R_1:L']

    def remaining_constraints(self):
        """
        Return list of functions which return boolean True/False values, True if
        the constraint is satisfied, False otherwise. Inputs to functions are,
        in order: decision, energy_vars, resource_vars, state
        """

        def iec1(state,decision,nodes):
            return decision['E:L']+decision['E:R_1']<=nodes['E'].get_preds_value(state)
        def iec2(state,decision,nodes):
            return decision['R_1:L']<=nodes['R_1'].get_preds_value(state)
        def iec3(state,decision,nodes):
            return decision['G:R_1']>=-(nodes['R_1'].get_preds_value(state))
        def iec4(state,decision,nodes):
            return decision['G:L']>=0.0
        def iec5(state,decision,nodes):
            return decision['E:L']>=0.0
        def iec6(state,decision,nodes):
            return decision['E:R_1']>=0.0
        def iec7(state,decision,nodes):
            return decision['R_1:L']>=0.0

        Inequality_Constraints=[iec1,iec2,iec3,iec4,iec5,iec6,iec7]

        return Inequality_Constraints

    def reward_function(self):
        """
        Return a function, which given the current state and an action, will
        return a reward. Inputs to function are, in order: state, decision,
        price_vars,energy_vars, resource_vars
        """
        def R(state, decision, nodes):
            return -1.0/1000*nodes['G'].get_preds_value(state)*(decision['G:R_1']+decision['G:L'])

        return R
```

Figure 4. Example_Constraints_and_Rewards.py

10. Filling in a row with the 'Action Space' Identifier

As with setting constraints and rewards, determining the feasible decision space is highly problem dependent as well. Set the 'Type' field such that the correct constraints and rewards file is read in for the problem (in Figure 3, we set it to 'Example Decision Space') and follow the instructions

outlined in the template 'custom_decision_space.py' to fill in the allowed_actions function. One way to find feasible decisions is to expand all possible (infeasible or feasible) decisions from a pre-decision state and use the constraints defined in constraints and rewards to determine if a decision is feasible. There is a built in script for this which can be read in by setting 'Type' to 'basic_decision_space.' However, in many problems this is too computationally expensive. Using knowledge of the problem, it is possible to choose fewer decisions to consider. For example, in the example problem, we might assume that the wind energy to load decision (E:L) should always be x_t^{EL}=min(E_t,L_t). This can be coded directly into the action space template to reduce time spent checking infeasible or bad decisions. Example logic for the example problem is shown below:

```
62
63      def allowed_actions(self, current_state):
64
65          '''
66          From the current state, return a list of feasible decisions.
67          '''
68
69          E_curr=self.Nodes['E'].get_preds_value(current_state)
70          L_curr=self.Nodes['L'].get_preds_value(current_state)
71          R_curr=self.Nodes['R_1'].get_preds_value(current_state)
72          Decision_Possibilities={}
73          Res1=self.Nodes['R_1']
74          d_int=Res1.get_discretization_interval()
75          R_max=Res1.get_max()
76          CR=Res1.max_charge_rate
77          Eff=Res1.conv_rate
78
79          E_to_L=min(E_curr,L_curr)
80          L_left=L_curr-E_to_L
81          E_left=E_curr-E_to_L
82
83          RL=[0.0,min(CR,R_curr,L_left)]
84          Decision_Possibilities['R_1:L']=RL
85          ER=min((R_max-R_curr), E_left, CR)
86          Unfilled_R=R_max-(R_curr+ER)
87          Decisions_1=[]
88          for rl in RL:
89              D={}
90              D['R_1:L']=rl
91              D['E:L']=E_to_L
92              D['E:R_1']=ER
93              Decisions_1.append(D)
94          #T5=timeit.default_timer()
95          #print T5-T4
96          decisions_forward=[]
97          for D in Decisions_1:
98              D['G:L']=L_left-Eff*D['R_1:L']
99              GRlist=set(np.arange(max(-(CR-D['R_1:L']),-(R_curr-D['R_1:L'])),mir
100             GRlist.add(0.0)
101             for GR1 in set(GRlist):
102                 DD=copy(D)
103                 DD['G:R_1']=GR1
104                 decisions_forward.append(DD)
105         return decisions_forward
106
            #T6=timeit.default_timer()
```

Figure 5. Example logic for determining feasible decisions.

11. Filling in a row with the 'Policy' Identifier:

Again, policies are highly problem dependent. However, there are 3 pre-defined policies for VFA-based policies that will read in value functions written in a specific way in a file whose name is placed in the FilePath field for either 1. Lookup table values for post-decision states ('Type'='VFA_Post_Lookup'), 2. Linear parametric belief model for post-decision states ('Type'='VFA_Post_Parametric', this also requires that the same basis functions be defined in both the solver and the policy script in the appropriate spots), or 3. Linear parametric belief model for pre-decision states ('Type'='VFA_Pre_Parametric', this also requires that the same basis functions be defined in both the solver and the policy script in the appropriate spots). These value functions are output from the MDP or ADP solver modules to be described later. There are also 5 blank policy classes, policy1,

policy2,…, policy5 left to be filled in according to a policy class template. To use policy4, for instance, enter 4 in the 'Type' field and enter any necessary parameters that accompany the policy in the 'Params' field, separated by a semicolon as always. If more than 5 policies types are to be tested at once, one must create a new policy class (copy one of the templates and rename), save the file as a new name, and import it into main, then add an additional 'elif' condition in the appropriate section of main directing the extra policy to be loaded in as well. For example, if one wants to compare 6 policies: create a class called policy6 in a new file and save as policy6; import policy6 into main.py; in the policy read-in section of main, add an elif(Type='6') statement to match the first 5 statements; and enter 6 in the 'Type' field of the input file, followed by the parameters. Policies that wish to be stored should be saved as a different name, and then changed back to policy1, … , or policy5 when used again. It is important to note that this is only necessary if one wants to created different classes or types of policies. If one wants to compare a policy with 10 different sets of parameters, only one policy is necessary. In this case, the Name field should be used to differentiate between the policies, and no two instances on any policy (or across all the policies) should have the same name. An example buy-low sell-high PFA policy is shown being read-in in the example.

12. *A note on 'Constraints_and_Rewards', 'Action_Space', and 'Policy'.*

Note that if a custom action space is defined, the constraints in constraints and rewards often do not need to be as the function is not called. Similarly, for PFA policies, which map a state to an action directly, it is often not necessary to define an action space or constraints. Other policies, such as VFAs which maximize the one-step contribution of an action plus the expected value of the downstream state usually require that an action space, and possibly constraints, be defined.

13. Filling in a row with the 'Module' Identifier

Finally, the module commands will run the program. Unlike the previous identifiers, the order in which these are entered in the spreadsheet will affect the order in which they are run. Thus, if one wants to tune the parameters of a policy, and then compare the policy to others, the parameter tuning module must be entered above the policy testing module in the spreadsheet.

Entering Type 'Custom' will allow the user to run any custom modules/functions that the user has created in the appropriate spot in the 'main.py' file (near the end of the script).

Entering Type 'Policy Compare' will test all policies defined in the spreadsheet on a specified number of sample paths. The output text file name should be placed in the FilePath category. 5 parameters are to be entered in the Params slot, separated by semicolons. The first is the third the number of sample paths to test each policy on an integer. The second parameter is a Python style list containing the names of all nodes to be plotted at each time step. The third parameter is a Python style list of the same length as the second parameter. This list consists 0/1 entries indicating whether the node in the corresponding position in the previous list is to be plotted in an average value per time period per policy sense (enter 0 for this option) or if the value for each sample path and policy is to be plotted (enter 1 for this option). For example, in the first example, 'E' is plotted individually, while, 'R_1', 'L', and 'G' are plotted on average. The individual option should only be chosen if the number of trials is small such that graphs do not become too cluttered. The fourth parameter is a Python style list containing the names of all decision variables to be plotted. The fifth parameter is the same as the third

parameter, except determines the plotting characteristics of the variables named in the fifth parameter. The eighth parameter is a Python style list containing the names of all the flow/decision variables to be plotted. For parameters 2 through 5, something must be entered, even if plotting is not desired, or plotting is not desired for a certain class of variable, thus enter the empty list [] for parameters 4 through 5 to skip plotting. The reward at each time step, and the cumulative reward at each time step will be plotted on average automatically for each policy.

Entering Type 'Exact Backward DP' will solve the full MDP and output values for each post-decision state at each time t in the text file specified by FilePath. This output can then be used with the 'VFA_Lookup' policy to test the policy (as long as the other aspects of the problem remain the same). Given a system is in a certain state, the optimal decision is found by maximizing the sum of the reward gained by the decision plus the value of the post decision state that results from the decision. The sole parameter that must entered is the number of time steps in the MDP.

Entering Type 'Backward ADP Post-Decision Lookup Table' will use backward ADP algorithm #1 from the 'Durante_Energy_Storage_Backward_ADP_10_10_17' paper in the docs folder to find lookup table VFAs for each post-decision state at each time t. These are output in the text file specified by FilePath. This output can then be used with the 'VFA_Post_Lookup' policy to test the policy (as long as the other aspects of the problem remain the same). Two parameters must be entered, separated by a semicolon. The sole parameter that must be entered is the sample rate for the pre-decision states. The smaller the sample probability, the quicker the run time, but the less accurate the results will be. The lower bound on the number of pre-decision states sampled in this implementation is the number of post decision states in the previous time period, as we require that each post decision state can transition to one subsequent pre decision state that has been sampled.

Entering Type 'Backward ADP Pre-Decision Linear Parametric' will use backward ADP algorithm #2 from the 'Durante_Energy_Storage_Backward_ADP_10_10_17' paper in the docs folder to find linear (in the parameters) VFAs for each pre-decision states at each time t. To use this module, one must enter basis functions extracting features of the state variable in the file 'backward_ADP_solver_Pre_Decision_State_Lin_VFA.py' in the space reserved for the entry of basis functions. The value for each of the sampled states is calculated, and then, using ordinary least squares regression, the parameters for each basis function at time t are determined. Using these best-fit parameters and the basis functions, the value for pre-decision states is approximated and stored for use in finding the values of the previous post-decision states. The low-dimensional VFA at each time t is then output in the text file specified by FilePath. This output can then be used with the 'VFA_Pre_Parametric' policy to test the policy (as long as the other aspects of the problem remain the same). The sole parameter that must be entered is the sample rate for the pre-decision states.

Entering Type 'Backward ADP Post-Decision Linear Parametric' will use a backward ADP algorithm to find linear (in the parameters) VFAs for post-decision states at each time t. To use this module, one must enter basis functions extracting features of the state variable in the file 'backward_ADP_solver_Post_Decision_State_Lin_VFA.py' in the space reserved for the entry of basis functions. For a pre-specified number of Monte Carlo samples the following happens: a time t post decision state is sampled, a random transition is made from post to pre decision state, and then using the time t+1 VFA for time t+1 post-decision states, a sample realization of the value of the sampled time t post-decision state is the maximum of the one-step contribution plus the approximate value of the

downstream time t+1 post-decision state. Then based on these sample realizations and the basis functions evaluations for the sampled time t post-decision states, ordinary least squares regression is used to determine the best fit parameter vector at time t to form the time t VFA. The low-dimensional VFA at each time t is then output in the text file specified by FilePath. This output can then be used with the 'VFA_Post_Parametric' policy to test the policy (as long as the other aspects of the problem remain the same). The sole parameter that must be enetered is the number of Monte Carlo samples per time step.

Entering Type 'Forward ADP API' will use a forward ADP algorithm (Approximate Policy Iteration, see the 'Durante_Energy_Storage_Backward_ADP_10_10_17' paper in the docs folder and read the section on forward ADP in the numerical results, policies tested section, refers to a Daniel Jiang paper) to find linear (in the parameters) VFAs for post-decision states at each time t. To use this module, one must enter basis functions extracting features of the state variable in the file 'forward_ADP_solver_linear_parametric_API.py' in the space reserved for the entry of basis functions. The value for each of the sampled states is calculated, and then, using ordinary least squares regression, the parameters for each basis function at time t are determined. Using these best-fit parameters and the basis functions, the value for each post-decision state is approximated. The low-dimensional VFA at each time t is then output in the text file specified by FilePath. This output can then be used with the 'VFA_Post_Parametric' policy to test the policy (as long as the other aspects of the problem remain the same). Two parameters must be entered, separated by a semi-colon. The first is the number of forward simulations to run between policy (parameter vector) updates (usually larger), and the second is the number of total policy (parameter vector) updates (usually smaller).

More Modules to be added… plus the following are modules from a previous version of the package that are not yet compatible with the new version:

Entering Type '5' will run a module intended to optimize parameters of the system (for example, the capacity of the resource) given a fixed control policy. The algorithm uses Thompson sampling, after an initial period where all system parameter possibilities are sampled n times (n specified by user), to determine which parameter values to test next. Up to 3 or 4 dimensions of a single component can be optimized at once; more than that is likely too many dimensions to handle. FilePath should contain the name of the output file that results will be written to. The results will contain a summary of cumulative rewards realized from all system parameter combinations tested. The chosen parameters are from the combination with the highest expected cumulative reward. The Params field should contain 9 entries separated by semicolons. The first entry is the start time of the energy storage control problem, and the second entry is the number of time periods over which we will be accumulating rewards. The third entry is the name of the fixed policy to be used. The policy must be created earlier in the spreadsheet. The fourth parameter is the name of the component for which we are optimizing system parameters. The component must have been previously created in the spreadsheet. When it was created, the component should have required specific input parameters in specific positions. To pick which of these parameters to optimize, in the fifth parameter here, enter a python style list containing the positions of the system parameter to optimize, where indexing begins at 0. For example, if the component required input parameters a; b; c and we want to optimize over a and c, we would input the list [0,2] for the fifth parameter. The sixth parameter is a list of the same size as the fifth parameter where each element is a two element list giving [minimum value, maximum value] for parameters that we want to optimize. Continuing the example, if we wanted to optimize a over the range k1 to k2 and c over the range d1 to

d2, we would enter in the sixth parameter: [[k1,k2],[d1,d2]]. The seventh parameter is a list of the same size as the fifth parameter where each element represents how many evenly spaced values of the system parameter we want to consider. Continuing the example, if we want to consider parameter a at the values k1, k1+(k2-k1)/3, k1+2(k2-k1)/3, k2 and parameter c at the values d1, d1+(d2-d1)/2, d2, we would enter the list [4,3] in the seventh parameter. The eighth parameter specifies how many experiments we run using Thompson sampling after the initial testing period. The ninth parameter specifies how many times we run a simulation with each combination of system parameters before the Thompson sampling period. For an example on how to use this module see Example User Input 4.

Entering Type '6' will run a module intended to optimize parameters of a control policy for the system given a certain fixed system configuration. The algorithm uses Thompson sampling, after an initial period where all system parameter possibilities are sampled n times (n specified by user), to determine which parameter values to test next. Up to 3 or 4 parameter of a single policy (if there are that many) can be optimized at once; more than that is likely too many dimensions to handle. FilePath should contain the name of the output file that results will be written to. The results will contain a summary of cumulative rewards realized from all policy parameter combinations tested. The chosen parameters are from the combination with the highest expected cumulative reward. The Params field should contain 8 entries separated by semicolons. The first entry is the start time of the energy storage control problem, and the second entry is the number of time periods over which we will be accumulating rewards. The third parameter is the name of the policy for which we are optimizing system parameters. The policy must have been previously created in the spreadsheet. When it was created, the policy should have required specific input parameters in specific positions. To pick which of these parameters to optimize, in the fourth parameter here, enter a python style list containing the positions of the system parameter to optimize, where indexing begins at 0. For example, if the policy required input parameters a; b; c and we want to optimize over a and c, we would input the list [0,2] for the fourth parameter. The fifth parameter is a list of the same size as the fourth parameter where each element is a two element list giving [minimum value, maximum value] for parameters that we want to optimize. Continuing the example, if we wanted to optimize a over the range k1 to k2 and c over the range d1 to d2, we would enter in the fifth parameter: [[k1,k2],[d1,d2]]. The sixth parameter is a list of the same size as the fourth parameter where each element represents how many evenly spaced values of the policy parameter we want to consider. Continuing the example, if we want to consider parameter a at the values k1, k1+(k2-k1)/3, k1+2(k2-k1)/3, k2 and parameter c at the values d1, d1+(d2-d1)/2, d2, we would enter the list [4,3] in the sixth parameter. The seventh parameter specifies how many experiments we run using Thompson sampling after the initial testing period. The eighth parameter specifies how many times we run a simulation with each combination of system parameters before the Thompson sampling period. For an example on how to use this module see Example User Input 4.

14. Useful Instructions for Creation of Custom/New Features:

Class Capabilities: The easiest way to view what each class is capable of doing is to view the templates where the class's mandatory functions are easy to view with the descriptions right beneath.

Global Variables: Passing the GLB_VARS class into a custom feature is an easy way to access all the variables that have been defined up to a certain point in the read-in process. Custom objects (classes, variables) may also be added to GLB_VARS by using GLB_VARS.set_global_variable(name, object) and

the value can be accessed later with GLB_VARS.get_global_variable(name), though once passed into a class, the GLB_VARS object must be defined in that class (usually done with self.GLB_VARS=GLB_VARS upon class initialization). Note that these are not Python global variables, but rather it is a class with a dictionary storing objects that is passed into most classes in this package. The default global variables names set by the default main.py program are:

- Horizon – the number of time steps in the optimization problem (an int)
- Discount_Factor – contains discount factor as a float
- Node – contains a dictionary of all nodes
- Decision_Vars – contains a list of all decision variable names
- Reward_Function – contains the reward function defined previously
- Equality_Constraints – contains the equality constraint function
- Equality_Decision_Variables – contains a list of the decision variables used in the equality constraint function
- Inequality_Constraints – contains a list of the functions created to enforce any inequality constraints
- Decision_Space – contains the Action Space formed by the input options or the user
- Policies – contains the policies defined in the spreadsheet

State Variable Form: The state variable is a Python dictionary. The value corresponding to the key 'node_name_x' is the time t state (physical, information, or belief state as applicable) of each node/process.

Decision Variable Form: The decision variable is a dictionary with each decision variable name as a key and the action to take as the value.