

Castle Labs

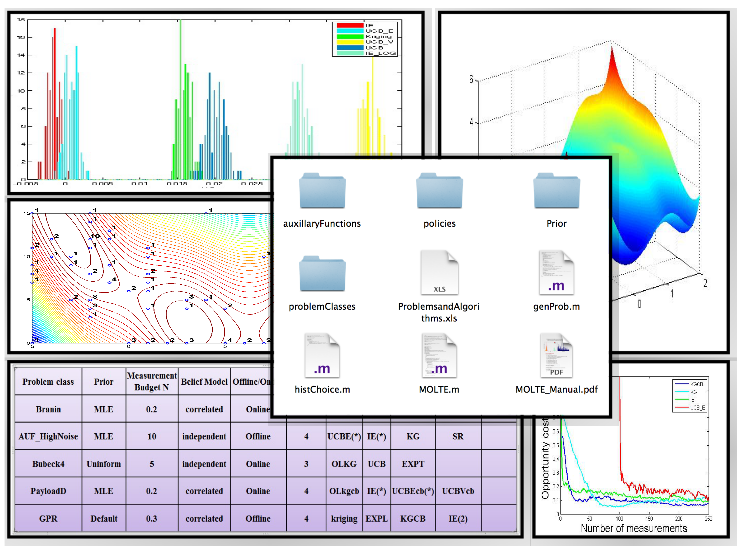
**Department of Operations Research and Financial Engineering**

**Princeton University**

**MOLTE**

**Modular, Optimal Learning Testing Environment**

August 12, 2017

****

Contents

[1. Introduction 1](#_Toc490591465)

[2. MOLTE-DF – Derivative-free stochastic search 2](#_Toc490591466)

[2.1. Description 3](#_Toc490591467)

[2.2. Construction 4](#_Toc490591468)

[2.3. Input Arguments 4](#_Toc490591469)

[2.4. Output Data and Figures 5](#_Toc490591470)

[2.5. Adding in new problem classes and/or policies 7](#_Toc490591471)

[2.6. Pre-coded Problem Classes 7](#_Toc490591472)

[2.7. Pre-coded Policies 8](#_Toc490591473)

[3. MOLTE-DB – Derivative-based stochastic search 9](#_Toc490591474)

[3.1. Description 10](#_Toc490591475)

[3.2. Input Arguments 11](#_Toc490591476)

[3.3. Supported problem classes 11](#_Toc490591477)

[3.3.1. Newsvendor 11](#_Toc490591478)

[3.3.2. Maximum Likelihood Estimation (linear model) 11](#_Toc490591479)

[3.3.3. Energy inventory problems 12](#_Toc490591480)

[3.4. Outputs 12](#_Toc490591481)

[4. MOLTE-MCTS 13](#_Toc490591482)

[4.1. Description 13](#_Toc490591483)

[4.2. Construction and Dependencies 14](#_Toc490591484)

[4.3. Input Arguments 15](#_Toc490591485)

[4.4. Output Data and Figures 15](#_Toc490591486)

[4.5. Problem Customization 16](#_Toc490591487)

[4.6. SimPolicy\_ define the value of a node and write up a rollout policy 16](#_Toc490591488)

[5. MOLTE-ADP 17](#_Toc490591489)

[5.1. Description 17](#_Toc490591490)

[5.2. Construction 18](#_Toc490591491)

[5.3. Input Arguments 18](#_Toc490591492)

[5.4. Output data and figures 18](#_Toc490591493)

[5.5. 1 dimensional state variable - inventory is variable 20](#_Toc490591494)

[5.5.1. BackwardsMDP 20](#_Toc490591495)

[5.5.2. Backwards ADP (simple -not pre and post- decision states)`- implementation issues 21](#_Toc490591496)

[5.6. 2 dimensional state variable - inventory and price are variable 21](#_Toc490591497)

[5.6.1. Backwards ADP (simple -not pre and post- decision states) - implementation issues 21](#_Toc490591498)

[5.7. 3 dimensional state variable - inventory and price are variable 22](#_Toc490591499)

[5.7.1. Backwards ADP (simple -not pre and post- decision states) - implementation issues 22](#_Toc490591500)

[5.7.2. Forward ADP (single pass) - implementation issues 22](#_Toc490591501)

[5.8. EXERCISES 23](#_Toc490591502)

[5.9. Solutions 23](#_Toc490591503)

# Introduction

MOLTE was originally envisioned as a flexible testing environment for comparing learning algorithms for what are often known as multi-armed bandit problems. The motivation was to serve a community which would develop new methods for solving these problems, and then evaluate them primarily using theoretical bounds or convergence rates. Computational testing was generally quite weak.

The architecture of the original MOLTE was designed by Yingfei Wang, a Ph.D. student in Computer Science at Princeton University, under the supervision of Warren Powell, a professor in the Department of Operations Research and Financial Engineering at Princeton. Yingfei set up a Matlab-based environment which enabled the creation of libraries of problems and policies, each encoded in its own Matlab.m file. Then, a spreadsheet interface allowed the user to specify which policies would be tested, and which problems they would be applied to.

This architecture makes it possible for researchers to add new policies and then test on an extensive library of problems. New problems are just as easy to add.

In 2017, three new modules were added to the MOLTE library, which now includes:

* MOLTE-DF – Derivative-free stochastic optimization
* MOLTE-DB – Derivative-based stochastic optimization
* MOLTE-MCTS – A general purpose library for Monte Carlo tree search
* MOLTE-ADP – A library for comparing a number of variations of forward and backward approximate dynamic programming, currently for general inventory problems.

These modules were written by different students, guided by the original MOLTE system (now called MOLTE-DF) designed by Yingfei Wang.

The MOLTE suite can be used by itself, but we are writing it as a companion to our new book, Optimization under Uncertainty: A Unified Framework, which can be downloaded from

<http://castlelab.princeton.edu/ORF544.htm>

This software is being offered in the public domain for the benefits of the research community. You are welcome to use it for any purpose. We ask only that if you add new modules (whether they are problems or policies), please send these to Warren Powell ([powell@princeton.edu](mailto:powell@princeton.edu)).

# MOLTE-DF – Derivative-free stochastic search

Assume that we have a function (possibly unknown, but not necessarily) that depends on a controllable parameter *x* (which can be a vector), where *x* can take on one of a set of discrete values . Our goal is to solve

 (2.1)

where  is a random variable, with a possibly unknown distribution. All we assume is that we cannot compute , but we can make noisy observations of . Since *x* is discrete, it is often useful to assume that there is an unknown truth  where

.

The goal is to find a *policy*, sometimes known as a decision rule (and in some cases, an algorithm) that determines how to go about learning . Let  be our state of knowledge about . We assume that our policy, that we designate by  returns a choice  which we then use for the  experiment which produces an observation  which we view as a noisy observation of . We then use this observation to update our belief about  (or ).

While we aspire to solve (2.1) above, this would mean finding an optimal solution . We refer to this as the *asymptotic formulation* of (2.1). In practice, we are going to design a sequential learning algorithm where we assume we have a budget of *N* experiments. After we have exhausted our budget, we obtain a solution that we call  to reflect both the policy used to find it, and the budget. The solution  depends on what we observe while we are running our experiments, which arises from the stochastic process

.

This means that if we were to repeat the process again (or run it in parallel) we would obtain a different answer, which tells us that  is a random variable.

We take a Bayesian approach, where  captures an initial distribution of belief about

 for all . For example, we may assume that  where  is the prior mean and  is the prior *precision*, where . We may assume that our belief about  is independent of , but we are going to allow different belief models from one problem to the next. Our goal is to find the best policy for finding . There are two objective functions we might use. The first is to maximize just the performance of the solution after all the experiments have been run, an objective that we call the *terminal reward.* This problem would be written

. (2.2)

Here,  is the random variable used to evaluate , which is not to be confused with the sequence  which were observed during the learning process.

The second objective arises when we have to experience the results of each experiment, such as what would happen if we were trying to find the best price for selling a product in an online marketplace. Here we are looking for a policy that solves

. (2.3)

Here, we want a policy that learns as we progress, maximizing returns as we go.

The problem in (2.3) is famously known as the *multiarmed bandit problem*, while (2.2) has been widely studied under the name *ranking and selection problem.* In reality, both of these are just derivative-free stochastic optimization problems, with two different types of objectives that we refer to as *terminal reward* (equation (2.2)) and *cumulative reward* (equation (2.3)).

The cumulative reward version (2.3) requires designing a policy that trades off between maximizing rewards each time period (“earning”) and learning better estimates  so that we make better decisions in the future (“learning”). This is known as the “exploration vs. exploitation” tradeoff. However, this tradeoff actually exists for both the terminal reward (2.2) and cumulative reward (2.3) objective functions.

MOLTE-DF is designed to make it possible to compare a wide range of learning policies on a wide range of learning problems. The user is allowed to choose the measurement budget, the noise involved in observing , and the choice of objective (equation (2.2) or (2.3)).

The remainder of this section describes MOLTE-DF and how it works.

## Description

MOLTE-DF is a sequential design-of-experiments (stochastic optimization) testing environment for testing learning algorithms on a wide range of offline and online problems. The Matlab-based simulator allows the comparison of a number of learning policies (represented as a series of .m modules) in the context of a wide range of problems (each represented in its own .m module). The choice of problems and policies is guided through a spreadsheet-based interface. Users can follow the standard APIs to define a new problem class and new policy by writing a separate .m file.

## Construction

MOLTE.m compares the polices specified in the Excel spreadsheet for each problem class for numP=100 times (which can be modified in MOLTE.m). Each time the simulator is run, it generates numTruth (which can be modified in MOLTE.m) different sample paths, shared between all the policies, computes the value of the objective function for each sample path and then averages the numTruth trials as the expected final reward or the expected cumulative rewards. The user may select to evaluate policies using either an online (“bandit”) objective function, or an offline objective function (ranking and selection, stochastic search).

## Input Arguments

Spreadsheet: an Excel file (ProblemsandAlgorithms.xls) with each row a problem class with the specified policies under comparison. A possible spreadsheet is as follows:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Problem class | Prior | Measurement Budget | Belief Model | Offline/Online | Number of Policies | | | | |
| PayloadD | MLE | 0.2 | independent | Offline | 4 | Kriging | EXPL | IE(1.7) | ThompsonSampling |
| Branin | MLE | 10 | correlated | Online | 4 | OLkgcb | UCBEcb(\*) | IE(2) | BayesUCB |
| Bubeck4 | uninformative | 5 | independent | Online | 4 | OLKG | UCB | SR | UCBV |
| GPR | Default | 0.3 | correlated | Offline | 4 | Kriging | kgcb | IE(\*) | EXPT |

For each problem, the following information has to be provided

**Problem class** is the name of a pre-coded problem with a specified truth function, the number of alternatives and a default noise level. If it is a user defined problem, the user should write a .m file in the ‘problemClasses’ folder with the same name as presented in this spreadsheet. If the problem has parameters, the user can use ( ) after the class name to input parameters; Otherwise use the default parameter value. For example, GPR(50, 0.45;100) specifies the value of the parameter for Gaussian Process Regression. Specifically, the prior mean is drawn from N(0, \sqrt(sigma)), the covariance matrix is of the form sigma\*exp(-beta(x-x’)). M is the number of alternatives.



**Prior** indicates the ways to get a prior. **MLE** means using Latin hypercube designs and MLE for initial fit. **Default** can be used only for the problems (e.g. GPR and InanoparticleDesign) that have a default prior. **Given** means using the prior distribution provided by the user. It can be achieved either by specifying the parameters of the problem class, e.g. GPR(50, 0.45;100), or by providing a ‘Prior\_*problemClass*.mat’ file containing ‘mu\_0’, ‘covM’ and ‘beta\_W’ in the ‘Prior’ folder, e.g. Prior\_GPR.mat. **Uninformative** specifies zeros mean and infinite variance for each alternative.

**Measurement Budget** specifies the ratio between the time horizon of the decision making procedure to the number of alternatives, e.g. there are 100 alternatives in the pre-coded Branin problem and the time horizon is specified to be 5\*100 if this column is set to be 5.

**Belief Model** decides whether we are using independent or correlated beliefs for the policies which use a Bayesian belief model.

**Offline/Online** controls whether the objective is to maximize the expected final reward or the expected total rewards.

**Number of Policies** is the number of policies under comparison. This specifies the number of columns which contain the name of a policy to be tested, each represented in the corresponding .m file with the same name. If there are parentheses with a number after the name of the policy, it means setting the tunable parameter to the value specified in the parentheses. If there are parentheses with \*, it means tuning the parameters with respect to the entire table and using the tuned value in the comparison: Otherwise use the default value (in fact some policies, e.g. KG and Kriging, do not have tunable parameters).

## Output Data and Figures

All the data and figures are saved in a separate folder for each problem class. Within the folder of each problem class:

**objectiveFunction.mat** saves the value of the online or offline objective function achieved by each policy for each trial;

**choice.mat** saves the decisions made by each policy and the name of all policies;

**FinalFit.mat** saves the final estimate of the surface by each policy after the measurement budget exhausted, together with the corresponding truth.

**alpha.txt** saves the value of tunable parameter for each policy that requires tuning;

**offline\_hist.pdf** is the histogram for each policy describing the distribution of the expected final reward compared to the reward obtained by the reference policy (whichever policy that is listed as the first policy in the input spreadsheet);

**online\_hist.pdf** is the histogram describing the distribution of the expected total reward; e.g. the following left figure is obtained for online Bubeck4. A distribution centered around a positive value implies the policy underperforms KG;

**histChoice.m** can read in the **choice.mat** and generate the distribution of the chosen alternatives for each policy and each trial. e,g, the right figure shows the frequency of choosing each of the 100 alternatives with a measurement budget of 300.

**genProb.m** can read in the **objectiveFunction.mat** and depict the mean opportunity cost with error bars indicating the standard deviation of each policy (in OC\_hist.tif) as shown in figure 1 (left), together with the probability of each policy being optimal and being the best in figure 1 (right).

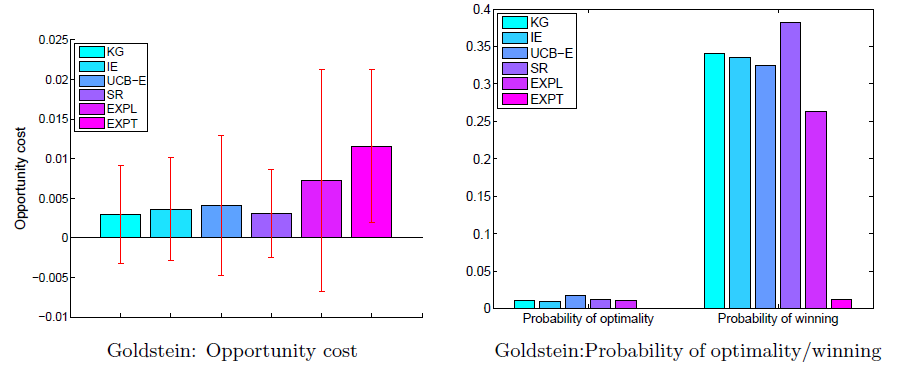


Figure -(Left) Opportunity cost for each policy. (Right) Probability each policy produces optimal solution, and probability each policy produces best solution.

The statistics stored in **objectiveFunction.mat, choice.mat** and **FinalFit.mat** can easily be used for other illustrations. For example, one can use the truth values stored in **FinalFit.mat** and the number of times each policy sample each alternative (sampling distribution) stored in **choice.mat** to generate the following two dimensional contour plot using Matlab commands contour (…) plot (…) and text (…), as well as the corresponding posterior contour using the final estimate of the surface stored in **FinalFit.mat**. The results are shown in figure 2.

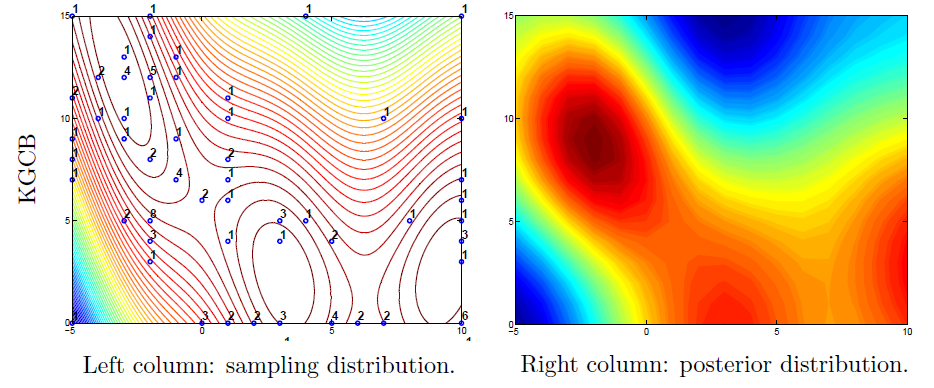


Figure 2-(Left) Countour plot with visitation counts. (Right) Heat map for posterior distribution.

Figure 3 shows the performance of each policy relative to the base policy.

## Adding in new problem classes and/or policies

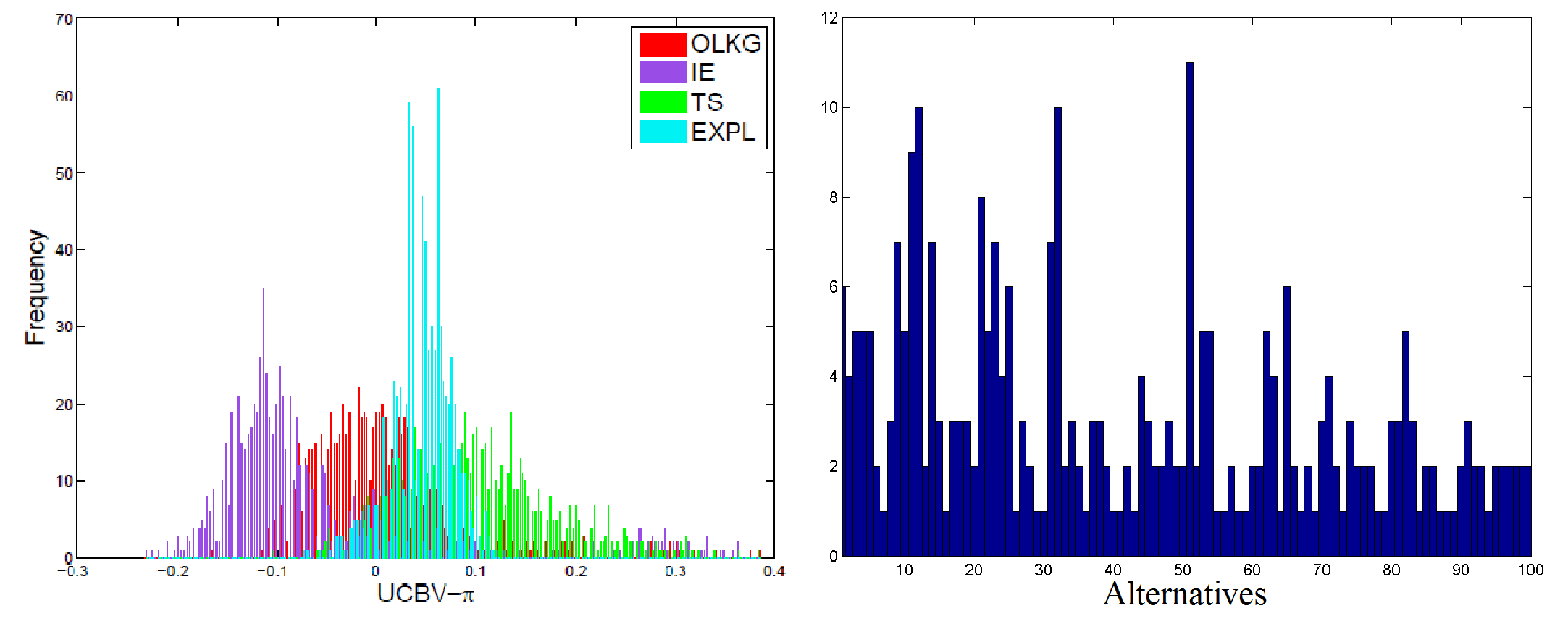


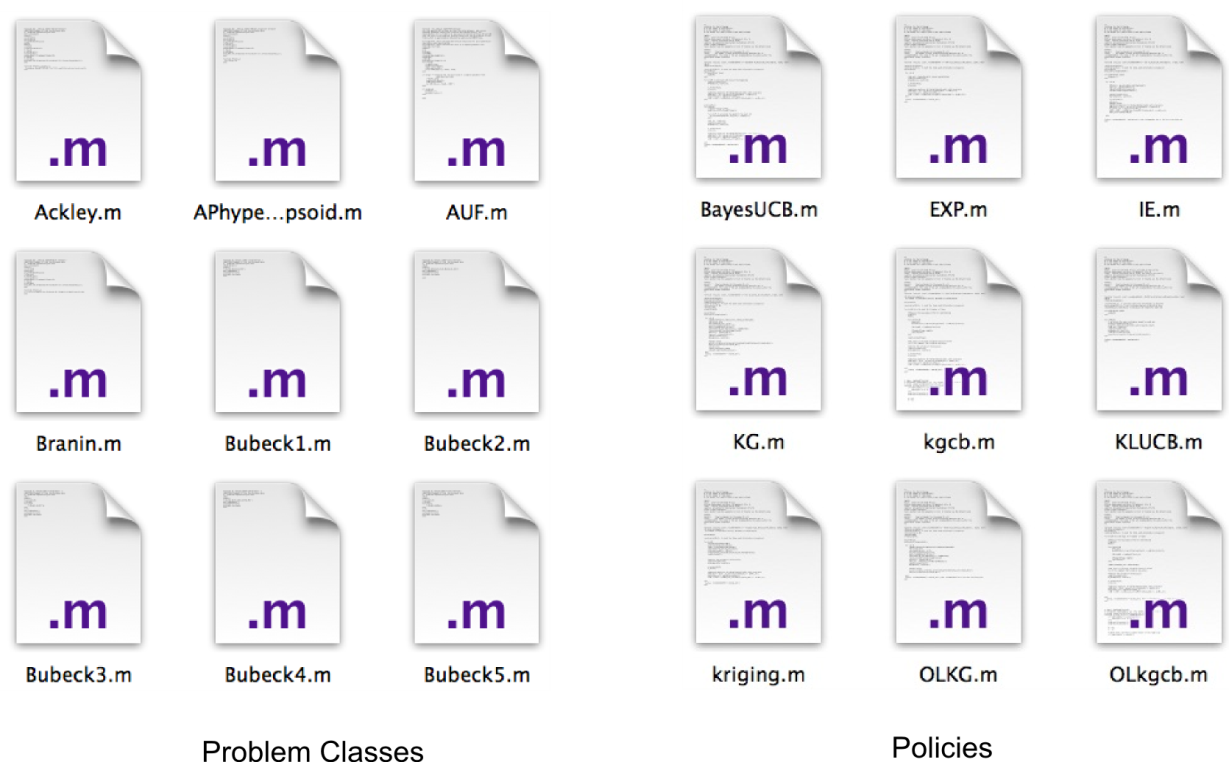
Figure -Offline-hist.pdf - (Left) Histograph of the performance of each policy relative to the base policy (positive means the policy outperforms the base policy). (Right) How often each alternative is tested by the reference policy.

**New problem class**

Each of the problems classes is organized in its own .m file in the ./problemClasses folder. The standard API is defined as Name(varargin), where varargin is used to pass input parameters with variable lengths for the problem class if needed.

**New policy**

Each of the problems classes is organized in its own .m file in the ./policies folder. The standard API is defined as KG( mu\_0,beta\_W,covM,samples, alpha, tune) where mu\_0 and covM is specifies the prior distribution, beta\_W is the known measure noise, samples are pre-generated and shared among all the policies, alpha is the tunable parameter and tune specifies whether tune this policy or use default value.



## Pre-coded Problem Classes

**Synthetic test functions:**

* Bubeck1~Bubeck7
* Asymmetric Unimodular Functions with the parameter chosen to be 0.2,0.5 and 0.8 with high or medium noise level, respectively
* Rosenbrock function with additive noise
* Pinter's function with additive noise
* Goldstein function with additive noise
* Griewank function with additive noise
* Branin’s function with additive noise
* Axis parallel hyper-ellipsoid function with additive noise
* Rastrigin’s function with additive noise
* Ackley’s function with additive noise
* Six-hump camel back function with additive noise
* Easom function with additive noise

**Truth-From-Prior experiments:**

* Gaussian Process Regression (with a default prior)

**Parameterized families**

* General AUF problems represented by one parameter drawn from U[0,1]

**Real world applications:**

* Payload delivery problem
* Immobilized nanoparticles design (with a default prior)

## Pre-coded Policies

* Interval Estimation (IE) (can be used for correlated beliefs)
* Kriging (can be used for correlated beliefs)
* UCB (and a modified version UCBcb incorporating correlated beliefs)
* UCBNormal
* UCB-E (and a modified version UCBEcb incorporating correlated beliefs)
* UCB-V(and a modified version UCBVcb incorporating correlated beliefs)
* Bayes-UCB (can be used for correlated beliefs)
* KL-UCB
* Knowledge gradient policy for offline learning (can be used for correlated beliefs)
* Knowledge gradient for online learning (can be used for correlated beliefs)
* Successive rejects
* Thompson sampling (can be used for correlated beliefs)
* Pure exploration (can be used for correlated beliefs)
* Pure exploitation (can be used for correlated beliefs)

# MOLTE-DB – Derivative-based stochastic search

MOLTE-DB starts with the same objective function as we did for derivative-free:

 (3.1)

The only difference is that we are going to assume that *x* is continuous and unconstrained. We could include constraints of the form  or box constraints of the form  without too much difficulty, but in its current form the algorithm does not allow constraints.

If  is concave for a given *W*, this means that  is concave. Now assume that we can compute a gradient (derivative if *x* is scalar)  for a given *W*. This is known as a stochastic gradient. For example, imagine we have a newsvendor problem

.

The stochastic gradient would be given by

.

MOLTE-DB implements the widely used algorithmic strategy of Robbins and Monro (1951), where (3.1) is solved using

, (3.2)

where  is a stepsize that determines how far we move in the directly of the stochastic gradient .

Stochastic gradients are easy to compute in many situations. The problem is determine an effective stepsize rule. For example, Robbins and Monro proved that (3.2) will asymptotically produce the optimal solution  to (3.1) if the stepsize satisfies the conditions



One rule that satisfies the conditions is , which can be the best stepsize in special situations. However, in most applications this works poorly. A major problem with the 1/n rule is that it tends to decrease too quickly. A simple variant is

, (3.3)

where  returns us to 1/n. Higher values of  produce stepsize sequences that decrease more slowly. This means we have to tune  to find the best performance.

As with the derivative-free problem in (2.1), we cannot actually run an algorithm an infinite number of iterations. Again assume that we have a budget of *N* iterations. Now, we are going to call our stepsize rule a “policy” since it is a rule for making the “decision” of which stepsize to use. In fact, our stochastic gradient algorithm is actually a dynamic system just as we encountered in the derivative-free problem. In this case, our state  is just the current solution . The decision is the stepsize  determined by our “stepsize policy” such as (3.3) (but we are going to consider many others). After making the “decision” , the stochastic gradient algorithm in (3.2) is our “transition function” that takes us to the next state . Let  designate our stepsize policy, and let  be the solution produced after *N* iterations. Our problem is to find the best stepsize policy, a problem that we can state as

. (3.4)

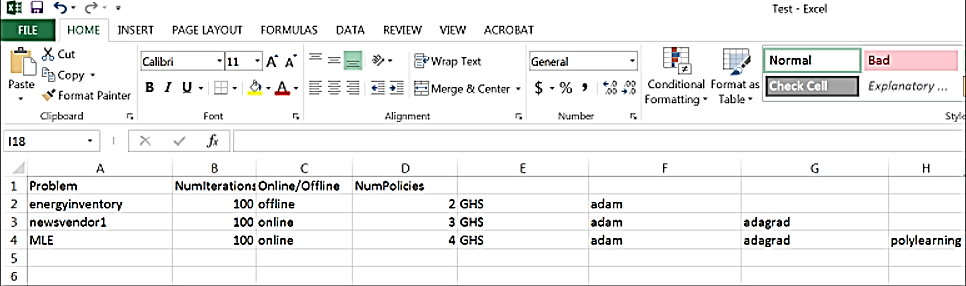
Anyone who has tried to use stochastic gradient methods has had to suffer the challenge of finding an effective stepsize rule (which is to say, to solve (3.4)). MOLTE-DB was created to allow students to experience the challenge of this experience, discovering in the process of why it is important.

The remainder of this section describes the MOLTE-DB system.

## Description

MOLTE-DB is a Matlab derivative-based stochastic optimization environment for testing a variety of stepsize policies on problems where the gradient can be computed or approximated. The simulator allows the comparison of stepwise policies (each represented in its own .m file) in the context of three problems (newsvendor, maximum likelihood estimation, and energy inventory). The user interacts with the interface through a Microsoft Excel spreadsheet, where problems and policies can be selected for testing. Users can define new problems and new policies by creating a .m file.

## Input Arguments



The user interacts with MOLTE-DB through an Excel spreadsheet called "DerivativeBased.xslx". The first column specifies the problem, the second column specifies online or offline (cumulative or terminal reward), the third column specifies the number of policies to test, and the next columns specify the names of the policies. The user can input parameters by typing in parentheses after the policy name. For example newsvendor(1,0.5,).

## Stepsize policies

The current set of supported stepsize policies are:

* GHS – Generalized harmonic, given by equation (3.3) above.
* adagrad
* adam
* BAKF – bias adjusted kalman filter
* kestens
* polylearning

Each policy is described in chapter 6 of *Optimization under Uncertainty*.

These policies are tested by simply entering one of these names in one of the columns E onward. Each of these policies are coded in a file called “stepsize.m” (for example, “adagrad.m”) in the folder “stepsize policies” in the MOLTE-DB folder. Recall that column E contains the reference policy, while the other columns contain the policies that are going to be compared against the reference policy.

## Supported problem classes

The optimum solutions for all these problem classes are obtained through the stochastic gradient algorithm given by equation (3.2).

### Newsvendor

Finding the optimal supply given a demand from an unknown random distribution and a desire to maximize profit (not overproduce such that supply is wasted or under-produce such that more demand could be fulfilled).

The following .m files are provided for variations of the newsvendor problem:

* newsvendor: demand is normally distributed, price is much larger than cost
* newsvendor1: demand normally distributed, price is close to cost
* newsvendor2: demand exponentially distributed price is much larger than cost
* newsvendor3: demand exponentially distributed price is close to cost

These files all output a vector of optimal supply estimates as well as vector of profits.

### Maximum Likelihood Estimation (linear model)

Find the parameters  for a linear model



using a stochastic gradient algorithm given a set of data. Note that for multiple parameters, our gradient is a vector of partial derivatives of our objective function with respect to each parameter.

The following .m files are provided for different MLE problems

* MLE: one parameter
* MLE1: two parameters

Note: The BAKF stepsize policy is not supported for this problem.

### Energy inventory problems

Given pricing data for battery storage, we address the problem of trying to find a sell price,  , and buy price, , such that profit is maximized. The program uses random restarts to initialize starting point for stochastic gradient algorithm since the problem is nonconvex. The default number of times we randomly restart is 5. The largest profit out of the five different starting points is saved.

Versions of the program include one that gets prices from real pricing data (energyinventory) and a program that obtains prices from a jump diffusion process (energyinventoryp1).

The following .m files are provided for different energy inventory problems.

* energyinventory: Prices are obtained from real pricing data.
* energyinventorp1: Prices are obtained from a jump diffusion process given by the following formula

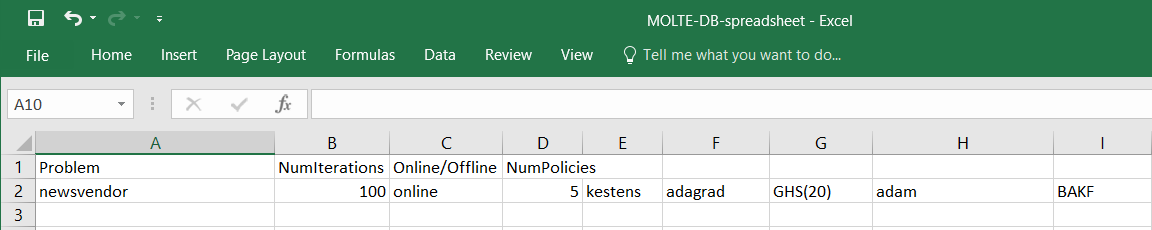


where  , , and  is an indicator variable that is 1 with probability  and 0 otherwise.

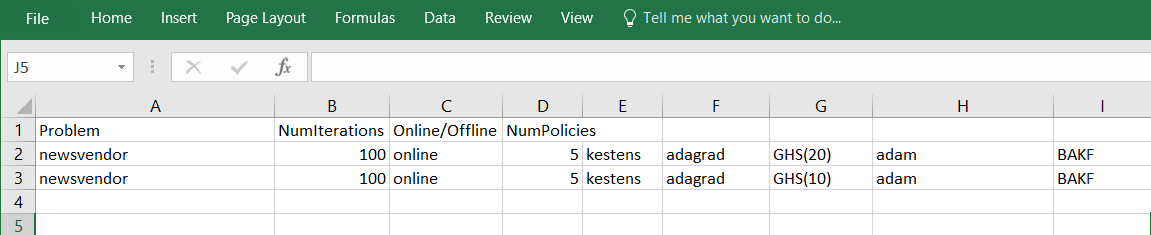
The stepsize rule BAKF is not supported for this problem.

## Running MOLTE-DB

Move to the “MOLTE-DB” folder within the MOLTE suite of Matlab programs. First bring up “MOLTE-DB-spreadsheet.xlsx.” Here you have to specify the experiments you want to run. The spreadsheet is initially set up to run “newsvendor” and should look like



Note that you can do more than one problem at a time. For example, you can run “newsvendor” and “newsvendor1” using:



## Outputs

MOLTE-DB produces a plot of “profits” or in the case of MLE, the mean squared errors (MSE). The first policy entered in the spreadsheet is the benchmark. Bars to the right of zero are “better” than the benchmark and bars to the left are “worse” than the benchmark (values are calculated as profit - referenceprofit). For all problem classes, positive is “better.”

Note that the parameter numPaths has been hardcoded into MOLTE-DB. For newsvendor and MLE problems, this parameter determines how many times a problem is simulated. If numPaths is 5, we will obtain 5 profits for each problem. For the sake of reasonably short runtimes, we only simulate the energy inventory policy one time for each stepsize policy.

# MOLTE-MCTS

Monte Carlo Tree Search (MCTS) replaces the explicit enumeration of the whole decision tree with a simulation policy to evaluate what might happen after we reach a state.

A state is characterized by

1. The pre-decision value function, the post-decision value function , and the contribution from being in state and taking action
2. The visit count , number of times we performed rollouts from
3. The count , number of times we took decision from
4. The set of actions at each state and the random outcomes that might happen in post-decision state

MCTS can be generalized into the following four components**:**

1. **Selection** Picks a decision and samples a subsequent outcome.
   1. **Actions** Chosen by a policy that controls the exploration-exploitation tradeoff. Here our default policy is the Upper Confidence bounding for Trees (UCT), which is written as follows:
   2. **Outcomes** Chosen via Monte Carlo sampling usually.

If an action (outcome, respectively) at a pre-decision state (post-decision state, respectively) has not been observed before, we perform 2). Otherwise, we move to the corresponding child node from the current node and perform 3).

1. **Expansion** Expands the Tree by generating a post-decision state (pre-decision state, respectively) node. At pre-decision state nodes, we get an estimate of the value of the current node by performing 3).
2. **Simulation** Obtains a quick and reasonable estimate of the value of being in a (pre-decision) state via a rollout policy.
3. **Backpropagation** Backtracks and updates the value of each states on the path leading to the current node.

## Description

MOLTE-MCTS is a Matlab-based optimizer that is pre-coded to tackle any of the following three problems:

1. **Traveling Salesman Problem** Suppose we have a set of cities **S**.Let be the policy that selects the next city to move to from the current city and be the realized cost associated with moving from to . Then we try to optimize the following:

where satisfies the following constraint:

where is the decision to go to city from city .

1. **Selling Problem** Suppose we are given an inventory , and forecasted pricesand fixed demands at time , and respectively. We aim to optimize the following:

where satisfies the following constraints:

And the following update equation:

where is the amount to sell at time .

1. **Pricing Problem**

## Construction and Dependencies

MCTSProblems.xls is the Excel interface to MOLTE-MCTS. It contains all the problem options available to the user, which are defined in the next section. MCTSInterface.m is the driver for the package – after configuring the desired the settings in the Excel file, simply open MCTSInterface.m in MATLAB and hit “run”.

The TSP problem requires a Java Runtime Environment, while the DeterministicIPSelling and DeterministicLPSelling policies for the Selling problem requires GUROBI. A get-away for the latter is to open the respective .m files and replace linprog\_GUROBI or intlinprog\_GUROBI with linprog and intlinprog respectively. However, do expect at least 30% slower runtimes if you opt to do so.

## Input Arguments

On the “Main” sheet of MCTSProblems.xls, the first seven columns are arguments that are **required** for each of the problems to run and the rest are problem-dependent and has default values selected if the user elects to leave it blank. The required options are listed as follows:

|  |  |  |
| --- | --- | --- |
| **Column** | **Description** | **Options** |
| Problem | The type of problems, see “Problems” sheet in Excel file | TSP, Selling, Pricing |
| Type\* | Stochastic or deterministic version of the problem | S for Stochastic, D for Deterministic |
| Rollout Policy | Policy used to evaluate value of a node | See “Policies” sheet in Excel file+ |
| d\_thr | Number of decisions to sample | Integer |
| e\_thr | Number of observations to sample | Integer |
| alpha | Parameter for UCT | Real Number (often small, e.g. 0 - 1.5) |
| Budget | Number of iterations per MCTS call in the problem | Integer |

\* Pricing problem is always stochastic regardless of option selected

The problem-specific options for Pricing is as follows:

|  |  |  |
| --- | --- | --- |
| **Column** | **Description** | **Options** |
| numExp | Number of experiments to run to learn the price | Integer |
| L | Number of steps to lookahead in rollout | Integer (less than numExp) |
| epsilon | Epsilon for epsilon-greedy rollout policy | [0 1] (0: exploitation, 1: exploration) |
| resolution | Size of discretization to compute differences between curves | The smaller, the higher resolution |
| weight | Determines the whether the value of the node is biased towards revenue or information gain | [0 1] (0:revenue, 1:information gain) |

The “Path” sheet of MCTSProblems.xls contains the path to the directory containing the following folders: “Pricing”, “Selling”, “TSP”, “Utilities”, and then following files: BackUp.m, MCTSInterface.m, MCTSProblems.m. The default setting is a relative path to the current directory (of MCTSInterface.m).

## Output Data and Figures

Returns a cell array that contains the results for each of the problems solved. For the TSP and Selling problems, the results struct contains the **bestSolution** - the results from MCTS, **optSolution** - the results from the deterministic problem, and **time** -the runtime.

The **MCTS** and **KGCBLin** fields from the results of the Pricing problem are solutions obtained from running those respective algorithms and **truth** stores the actual thetas used in the linear model. **time** is again, the runtime.

Similarly, the figures for TSP and Selling problems have the same structure. They contain the decisions made at each time t, the resulting revenue or cost, the cumulative revenue or cost, and the value at the root node of each tree for each time t – which allows us to look for convergence behavior. Figures resulting from the Pricing problem are similar with the exception that the value at root node plots are replaced by a plot showing qualitatively, how well we learned the price curve.

## Problem Customization

Because MCTS is a rather general methodology, we built this package with the mindset of allowing easy adaptation to different problems. The book-keeping is universal across problems and below are a list of changes you might have to make before deploying the package to your application:

**State\_** add/replace the state variables here.

**TreePolicy\_** adjust UCT to either “min” or “max”, and under **ExpandObservation()** and **ExpandAction()**, update the following:

**generateObservations\_**

**generateActions\_**

**transitionPre2Post\_**

**transitionPost2Pre\_**

## SimPolicy\_ define the value of a node and write up a rollout policy

# MOLTE-ADP

MOLTE-ADP uses the context of a simple inventory problem to illustrate a variety of dynamic programming algorithms. To illustrate the difference between this problem class versus those addressed by MOLTE-DF and MOLTE-DB, consider our basic newsvendor problem

 (5.1)

MOLTE-DF and MOLTE-DB present algorithms for pure learning problems, where we are trying to learn  (for the derivative-free version) or to learn the optimal solution  in the derivative-based version.

Now consider what happens when we make some minor alterations to this problem. The first is that we are going to assume that if we supply too much inventory, then the excess inventory, given by , is held over to the next period. We let  be the inventory available at the beginning of time *t*, which is what was left over from time *t-1*. We also make the assumption that all of  is available for sale to meet the demand , but we can add to this by ordering .

We can write this as a dynamical system where now our state variable is the inventory  we have available at time *t*, which means we would write . In MOLTE-ADP, we are going to assume that the demand  is discrete, which means that both  and  would be discrete as well. MOLTE-ADP uses discretized versions of the uniform and (truncated) normal distribution for demands.

## Dynamic programming

We can solve our inventory problem using Bellman’s equation, which is written

 (5.2)

This is what is generally meant when people talk about solving a problem using “dynamic programming.” Equation (5.2) is disarmingly simple, but computationally dangerous. We are going to explore two strategies for solving (5.2):

* Classical “backward” dynamic programming – When this works, it produces an optimal solution in the form of what is known as an optimal *policy*, which is a function for making decisions. Optimal policies are quite rare.
* Approximate dynamic programming. We will illustrate two styles of ADP:
  + Backward ADP
  + Forward ADP

The point of illustrating these tools is both to illustrate their power, as well as the challenges that arise when using them.

### BackwardsMDP

BackwardsMDP directly solves equation (5.2) which can be computed using

, (5.3)

where P(w) is the probability that the demand . We have to compute (5.3) for  over some specified range that reflects the smallest and largest values that  might take.

Solving (5.3) gives us the exact value functions , which then gives us an optimal policy

.

We can then simulate this policy forward in time by using  to find the order decision , after which we choose a random realization of  that leads us to the next state .

If we actually have a one-dimensional inventory problem, and we know the probability distribution , then backwards dynamic programming is the way to go. In practice, this is generally not the case. Instead, we are going to use our ability to solve one (and later two) dimensional problems exactly, and then use this to evaluate different approximation methods, which we can then use for more complex problems.

### Approximate dynamic programming

There are many settings where backward dynamic programming, using the standard “lookup table” representation of the value function (this means a value for each discrete state ) simply does not work. This would be the case if the state *s* was continuous, but most of the time we encounter problems when  is multidimensional, as we illustrate below.

Approximate dynamic programming replaces the exact value  with an approximation  which is our estimate after *n* iterations. We are going to consider two ways of representing our approximation:

* Lookup tables – Here, we assume that our state variable can be represented as a discrete value which we write as  (this is known as a “flat representation”). This means we need an estimate  for each . If  is multidimensional, then the number of discrete states can become quite large, which makes finding the approximation .
* Linear models – Here we assume that our value function has the structure



The functions  are known as *basis functions*, or *features.* These have to be designed by the analyst. Then, instead of estimating a value  for each state *s*, we just have to estimate the regression parameters  for .

There are different ways for estimating the values  (if we are using a lookup table representation) or the coefficients  when using a linear model. In this package we illustrate the following methods:

* + Backwards ADP – Here we step backwards in time as we would in computing equation (5.3), but instead of computing  for every state , we choose a sampled set of states , and then use these to create our approximation.
  + Forwards ADP – This is what is typically meant when people talk about “approximate dynamic programming.” This comes in two flavors: 1) Single pass and 2) Double pass.
    - **Single pass** – Single-pass forward ADP uses value function approximations  to create a policy



After computing , we then compute



which is a sampled estimate of being in state . We now use  to update  using (for the lookup table version):

.

If we are using a linear model, there are recursive equations for updating our estimate of the regression parameters  to obtain .

We then choose a sample  and use this to compute

,

and then repeat the process.

* + - **Double pass** – Here we step forward in time as we did with the single pass, but we do not update the value functions as we go. Instead, we just perform a full simulation, keeping track of the sequence of states , decisions , and . After we complete a full forward pass, we then perform a backward pass and compute



As we compute , we use this to update the value function approximation as we did with the single pass version.

## 1 dimensional state variable

We begin by considering a simple inventory problem where the state is given by the inventory . Bellman’s equation is then given by equation (5.2), with .

### Backwards ADP

For "big" values of T (time horizon) (usually for T>50), **BackwardsADP** provides "poor" profits (around 60 percent of the profit from **BackwardsMDP**). However, if we truncate the number of possible states for every time *t* from 1 to *T* when we compute the value functions, by eliminating the states with big inventories, we obtain profits better than 95 percent of the profits generated by **BackwardsMDP**. More exactly, for every time *t*, we only consider the inventories less than



where *boundx* is the upper bound for the possible actions that we can make (we assume that the lower bound is 1) and



## 2 dimensional state variable

inventory and price are variable



### Backwards ADP

\begin{itemize}

\item Choosing the basis functions for the linear model should be done very carefully since: \\

1. "Bad" basis functions produce "bad" results. \\

2. "Bad" basis functions produce a huge variance for both the profits obtained when we run the policy and for the mean of the profits when we run the policy.\\

\end{itemize}

Example(s) of "bad" basis functions:

\begin{itemize}

\item $1,i,i^2,i\*p,i\*p^2,p^2$ where $i$ is the inventory and $p$ is the price at a given state. In this case, \textbf{BackwardsADP} produces results around $40\%$ of \textbf{BackwardsMDP} for both independent prices and prices that depend on the previous price. However, sometimes, the mean of the profits can get as low as $17\%$ or as high as $95\%$.

\end{itemize}

The "good" basis functions: $1,i,i^2,i\*p,p,p^2$ , where $i$ is the inventory and $p$ is the price at a given state. In this case, \textbf{BackwardsADP} produces results better than $95\%$ of \textbf{BackwardsMDP} for both independent prices and prices that depend on the previous price.

## 3 dimensional state variable - inventory and price are variable

### Backwards ADP

Example(s) of "bad" basis functions:

\begin{itemize}

\item $1,i,i^2,i\*p,i\*p^2,p^2,x,x^2$ where $i$ is the inventory and $p$ is the price at a given state and $x$ is the decision made at time $t-1$ and implemented at time $t$. In this case, \textbf{BackwardsADP} produces results around $50\%$ of \textbf{BackwardsMDP} for both independent prices and prices that depend on the previous price. However, sometimes, the mean of the profits can get as low as $17\%$ or as high as $95\%$.

\end{itemize}

The "good" basis functions: $1,i,i^2,i\*p,p,p^2,x,x\*p,x\*i$ , where $i$ is the inventory and $p$ is the price at a given state and $x$ is the decision made at time $t-1$ and implemented at time $t$. In this case, \textbf{BackwardsADP} usually produces results better than $95\%$ of \textbf{BackwardsMDP} for both independent prices and prices that depend on the previous price. However, sometimes, it provides results around $80\%$ of \textbf{BackwardsMDP}.

### Forward ADP (single pass) - implementation issues

\begin{itemize}

\item The number of paths we simulate -$it$- plays a very important role if we keep all the other tunable parameters fixed. In other words, for every $it$ there are different parameters that work better.

\item For a relatively small $T$ (time horizon), if $it$ becomes too big the matrix of coefficient for our basis functions tends to $0$ and therefore the results are not accurate and hence relatively bad. For example, for $T=10$, $it$ should range between $500$ and $1000$.

\end{itemize}

## The software

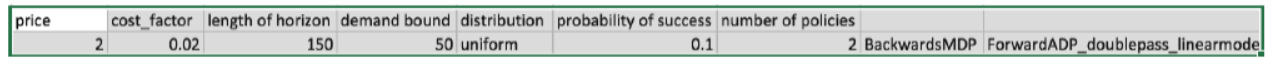
MOLTE-ADP is designed to allow students to study a variety of dynamic programming algorithms using the setting of single and multi-dimensional inventory problems. The package allows students to test classical backward dynamic programming, and compare the results to different approximation strategies using backward approximate dynamic programming (ADP) and forward ADP. Included are implementations using both lookup table belief models and (linear) parametric models. The user can guide the system through a standard Excel spreadsheet that makes it possible to specify which problems are being solved, and the algorithmic strategy.

### Description

Compilator.m compares the polices specified in the Excel spreadsheet for each problem class for *numP*=100 times (which can be modified in Compilator.m). Each time the simulator is run, it generates *numTruth* different sample paths (which can be modified in Compilator.m), shared between all the policies, computes the value of the objective function for each sample path, and then averages the *numTruth* trials as the expected final reward or the expected cumulative rewards.

### Input Arguments

Spreadsheet: an Excel file (*input\_data\_pbx.xls*) with the second row a problem class with the specified parameters and policies under comparison. A possible spreadsheet is as follows:



For each problem, there exists detailed explanations of all the parameters in the same Excel file where the input parameters are provided by the user.

However, in general, the parameters that can be provided are related to:

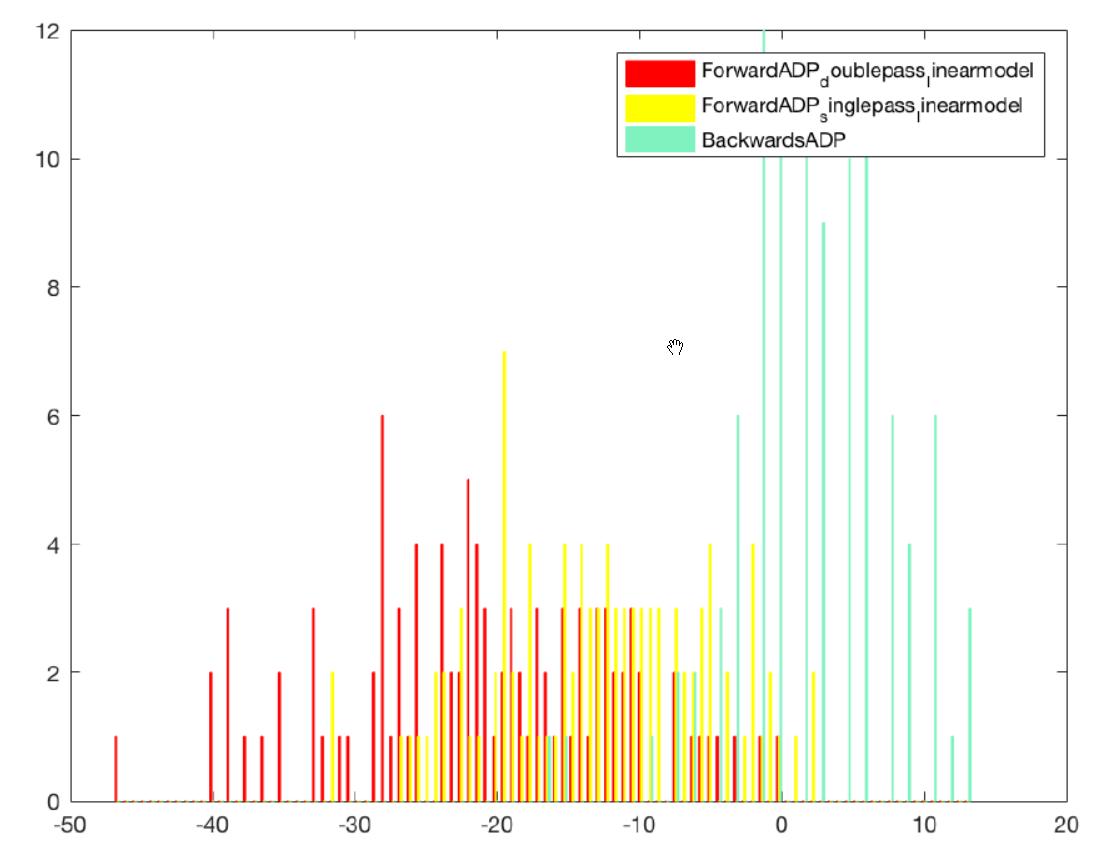
* Price  , which can be constant or follow certain distributions (uniform or normal)
* Dependence between consecutive prices - When the price follows a normal distribution, if the prices are independent, then for any time *t*, all the prices follow the same distribution. Else, if the prices are dependent, then the price at time *t+1* follows a normal distribution that has the mean equal to the price at time *t* and given standard deviation.
* Demand , which can follow either an uniform or binomial distribution with various lower and upper bounds.
* Length of horizon - *H*
* Cost  - which can be constant or depend on the demand at a given time by the formula .
* Number of policies - is the number of policies under comparison. This specifies the number of columns which contain the name of a policy to be tested, each represented in the corresponding .m file with the same name.

All policies are compared to the first policy listed.

### Output data and figures

Assume we have run our spreadsheet with inputs:



For every problem that we solve, we display a histogram of the performance of each policy minus the performance of the base policy. A sample result is given below

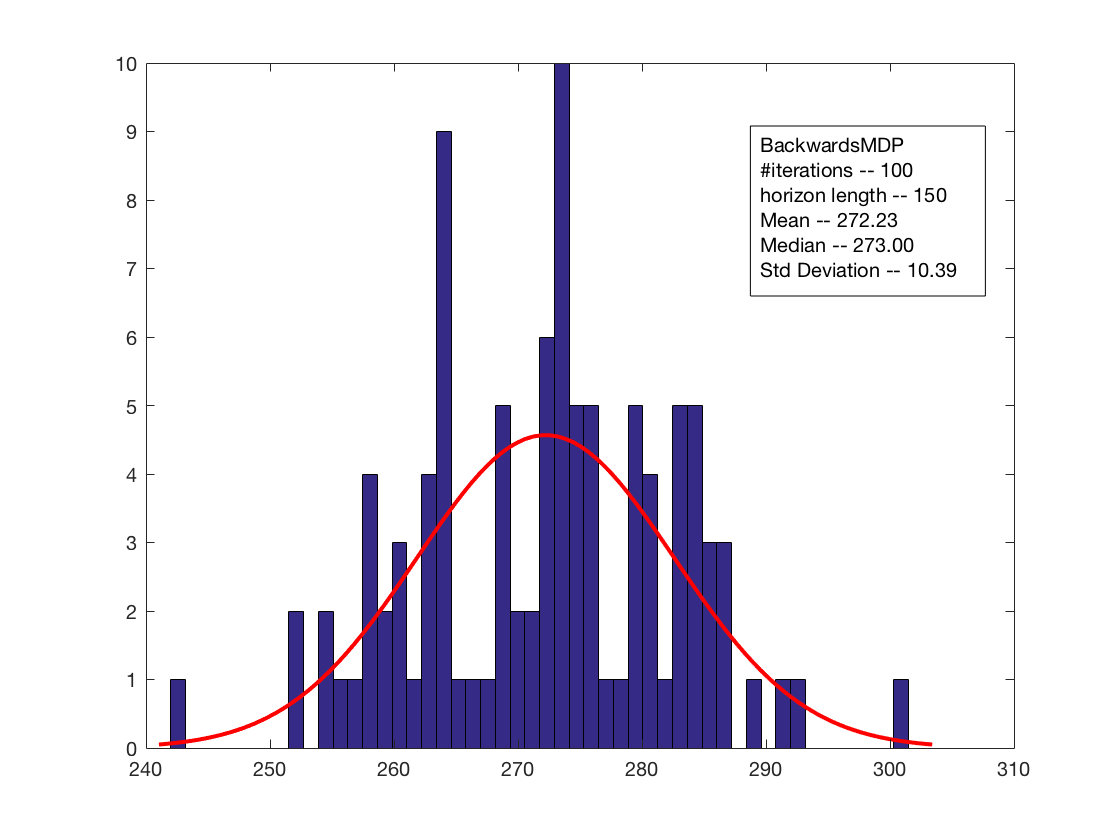
In our case, the first policy was BackwardsMDP, so the histogram is displaying results relative to BackwardsMDP, which is optimal.

Moreover, the software returns *numTruth* final results for each of our numTruth iterations for each policy (shown in the figure below). Moreover, it also provides the mean, median, and standard deviation of the results and a normal fit of the results (the red curve).

## EXERCISES

1. For a one dimensional problem, Problem 1, what are some "good" tuned values for the constant used in the stepsize and the explore probability formula when price , , demand bound=50, distribution=uniform for all the ForwardADP algorithms? Which of the algorithms works best when tuned?\\

2.For the two dimensional problem, Problem 2, what are some "good" tuned values for the constant used in the policy for the stepsize  and the explore probability formula  when price\_mean =25, price\_std=10, independent prices =1,fixed\_cost=5, variable\_cost=0.2, length of horizon =10, lower bound demand=1, upper bound demand =20, distribution=uniform, for all the ForwardADP algorithms? Which of the algorithms works best when tuned?

3. For three dimensional problem, Problem 3, what are some "good" tuned values for the constant used in alpha and the explore probability formula when price\_mean =25, price\_std = 10, independent prices =1, fixed\_cost=5, variable\_cost=0.2, length of horizon =10, lower bound demand = 20, upper bound demand =50, distribution=uniform for all the ForwardADP algorithms that use linear models? What is a "good" tuned value for the constant used in lambda (for the linear model algorithms)?

4. What is a good choice for the policies that use a linear model for the 2-dimensional state problems? See if you can achieve 95 percent of the optimal provided by BackwardMDP.

5. What is a good choice for the policies that use a linear model for the 3-dimensional state problems?

## Solutions

1.\\

$ForwardADP\\_singlepass\\_lookuptable$ \\

GOOD:(alpha, explore)=(50,10) \\

BAD:(alpha,explore)=(10,1) \\

$ForwardADP\\_singlepass\\_linearmodel$ \\

GOOD:(alpha, explore)=(150,50) \\

BAD:(alpha, explore)=(10,5) \\

$ForwardADP\\_doublepass\\_lookuptable$\\

GOOD:(alpha, explore)=(500,1)\\

BAD:(alpha, explore)=(1000,200)\\

$ForwardADP\\_doublepass\\_linearmodel$\\

GOOD:(alpha, explore)=(500,50)\\

BAD:(alpha, explore)=(10,1)\\

2.\\

$ForwardADP\\_singlepass\\_lookuptable$ \\

GOOD:(alpha, explore)=(50,15) \\

BAD:(alpha,explore)=(10,3) \\

$ForwardADP\\_singlepass\\_linearmodel$ \\

GOOD:(alpha, explore)=(40,20) \\

BAD:(alpha, explore)=(200,1000) \\

$ForwardADP\\_doublepass\\_lookuptable$\\

GOOD:(alpha, explore)=(10,3)\\

BAD:(alpha, explore)=(500,1)\\

$ForwardADP\\_doublepass\\_linearmodel$\\

GOOD:(alpha, explore)=(500,50)\\

BAD:(alpha, explore)=N/A \\

3. For $ForwardADP\\_singlepass\\_linearmodel$, it is important to realize that if we take lambda to be a constant (equal to $1$ for example), then the results are bad. A good choice for lambda would be $lambda(n)=\frac{500}{(500+n^{0.2})}$.

$ForwardADP\\_singlepass\\_linearmodel$ \\

GOOD:(alpha, explore)=(250,70) \\

BAD:(alpha, explore)=(10,5) \\

For $ForwardADP\\_doublepass\\_linearmode$l, we take $lambda(n)=1$. \\

$ForwardADP\\_doublepass\\_linearmodel$ \\

GOOD:(alpha, explore)=(1000,250) \\

BAD:(alpha, explore)=(1,250) \\

4.The "good" basis functions: $1,i,i^2,i\*p,p,p^2$ , where $i$ is the inventory and $p$ is the price at a given state. In this case, \textbf{BackwardsADP} produces results better than $95\%$ of \textbf{BackwardsMDP} for both independent prices and prices that depend on the previous price. \\

5.The "good" basis functions: $1,i,i^2,i\*p,p,p^2,x,x\*p,x\*i$ , where $i$ is the inventory and $p$ is the price at a given state and $x$ is the decision made at time $t-1$ and implemented at time $t$. In this case, \textbf{BackwardsADP} usually produces results better than $95\%$ of \textbf{BackwardsMDP} for both independent prices and prices that depend on the previous price. However, sometimes, it provides results around $80\%$ of \textbf{BackwardsMDP}. \\