MDPSOLVE
A MATLAB Toolbox for Solving Markov Decision Problems with Dynamic Programming

User's Guide

Paul L. Fackler*

June 6, 2011

*The author is a Professor in the Department of Agricultural and Resource Economics at North Carolina State University.

Mail: Paul L. Fackler
Department of Agricultural and Resource Economics
NCSU, Box 8109
Raleigh NC, 27695, USA

e-mail: paul_fackler@ncsu.edu

Web-site: http://www4.ncsu.edu/∼pfackler/

Contents

1 Notation Summary

A	action variable
$\mathcal{I}^x$	index representing the state associated with each state/action combination
n^a	number of values of the action variables (when this is the same for all states)
n^s	number of possible values of the state variables
n^x	number of possible values of the state/action combination
n^y	number of possible information signals in a POMDP
P	state transition probability matrix ($n^s \times n^x$)
Q	conditional probability of an information signal in a POMDP
R	one period reward ($n^x \times 1$ or $n^s \times n^a$)
S	state variable or matrix of state variable values (n^s-rows)
S^+	next period's state
T	terminal date in finite horizon problems
V	value function
X	state/action combination or matrix of state/action variable values (n^x-rows)
Y	an information signal in a POMDP
f	a state transition conditional probability density function $f(S^+\|X)$
g	a state transition equation $S^+ = g(X, e)$
e	a noise term in the state transition equation
w	the probability distribution associated with e

2 Introduction

MDPSOLVE is a toolbox of procedures written in the MATLAB programming language that facilitates the design and analysis of Markov decision problems (MDPs) using dynamic programming. This user's guide describes the features of the procedures and illustrates them with examples of environmental and resource management problems that can be addressed with dynamic programming.

The scope of the procedures in this toolbox are discrete time Markov decision problems that are characterized by (1) a finite set of discrete state variable values (the toolbox also contains procedures for working with continuous state variables), (2) a finite set of discrete action variable values, (3) a set of transition probabilities that describe the conditional probability of next period's state given this period's state and action, (4) a set of rewards, one for each state/action combination, that accrue to the decision maker and (5) a discount rate measuring the value of a unit of reward next period relative to a unit value in this period. The goal of a Markov decision problem is to find the optimal action to take for each value of the state and to determine the sum of the current and expected discounted future rewards.

This user's guide is not designed to teach users about MDPs and dynamic programming (many textbooks are available for this - see Section 19 for suggestions). A common notation, however, will be useful in describing the features of the software. It is assumed that at any point in time there are n^s values of the state variable S, denoted s_i, and n^x state/action combinations X, denoted x_j. To facilitate clarity of exposition, throughout this document, the subscripts

used for states and state/action combinations will always be i and j, respectively.

The first component of the problem is the objective. An n^x-vector R describes the rewards obtained for a given state/action combination. Specifically

$$R_j = \text{reward}(X = x_j)$$

These are the rewards that accrue each period. R should always contain n^x values, but can be specified as an $n^s \times n^a$ matrix when there are the same number (n^a) of possible actions in each state (`mdpsolve` has a wide variety of ways to specify problems; see Section 8 for details). In addition a discount factor must be specified that measures the value of a reward obtained in the next period relative to one obtained in the current period. The discount factor can equal 1 (see Section 10.3 for a discussion of the issues raised in this case). The objective is to maximize the sum of the current and the discounted expected future rewards up to some period T.

The second component of the problem is the model describing the dynamic behavior of the state variables. There are a number of ways that this can be described. The most basic uses an $n^s \times n^x$ transition probability matrix that defines the probability of obtaining each state in next period given the current period's state/action combination. Letting S^+ represent the state in the next period, an individual element of this matrix is

$$P_{ij} = Prob(S^+ = s_i | X = x_j)$$

P is known as a probability of stochastic matrix, which is a matrix with all non-negative values with each column summing to 1.[1]

An alternative way to define the dynamics of the state uses a transition function of the form

$$S^+ = g(X, e)$$

where e is a random variable with a specified probability distribution. The variable e is sometimes referred to as a shock or as environmental noise. Although MDPSOLVE assumes that the state dynamics are defined in terms of P, it is often more natural to use the functional representation. With a functional representation it is possible that S^+ is not one of the n^s values of the state variable (not one of the s_i) and some interpolation scheme is needed to convert this format into the probability matrix format (for further discussion see section 9).

In what follows installation procedures are discussed and a simple example is described and the code used for its solution is presented. Then basic considerations in specifying MDPs are discussed. This is followed by a section describing alternative ways to specify the transition probabilities and then by a section describing miscellaneous features. Then there is a section describing how to solve problems involving distinct stages and a section describing extensions to the basic model, including adaptive management, partially observed state models, and category count models. Finally a number of tools are described that facilitate analysis and interpretation of results.

[1] Technically this is a column stochastic matrix; if the rows summed to one it is a row stochastic matrix. For more discussion of this point see Section 5).

3 Installation

MDPSOLVE is distributed as a single compressed (ZIP) file. This should be extracted into its own directory. With some extraction utilities you will need to specify that the directory structure should be maintained (if you do not maintain the directory structure the code will still run but it will will be more disorganized and hence harder to use).

Extracting the files will create a main directory, `MDPSOLVE`, and four subdirectories, `mdptools`, `mdpdemos`, `mdputils` and `probability`. You may put the toolbox anywhere on your hard disk (or even a portable flash disk). It can also be placed in a new subdirectory of the MATLAB toolbox subdirectory (type `matlabroot` to determine where MATLAB is located on your machine). One reason for doing this is that procedures located in subdirectories of the MATLAB root directory run marginally faster (MATLAB assumes that they are unaltered and does not check for updates each time they are accessed) but it will also erase them when MATLAB is updated or reinstalled.

Regardless of where the files are located you will need to add the directories to the MATLAB path so they can be located by MATLAB. This can be done using the MATLAB path command or using the **File/Set Path** menu. To avoid doing this each time you start a session the path command can be put into a `startup` file that it is executed each time you begin a MATLAB session. In this way the capabilities of the toolbox are always available. A sample startup file is included in the MDPSOLVE directory (for more information on startup files type **doc startup** at the MATLAB command prompt). The sample file needs to be edited to reflect where the files reside on your machine.

All of the examples discussed in this document have script files located in the `mdpdemos` directory. These can be run by typing their name at the MATLAB command prompt and hitting **Enter** or by loading them into the MATLAB editor and clicking on the **Run** (green arrow) button (this requires that you have a relatively recent version of MATLAB). The basic toolbox functions are contained in the `mdptools` directory. The `mdputils` directory contains utility functions that are needed by the procedures in the `mdptools` directory; these procedures may be called by users but are separated from the main procedures to reduce file clutter.

Although all of the procedures in the toolbox are written in MATLAB there are also MEX versions of some of them that are coded in C. These must be compiled before they can be called by MATLAB (compiled versions for the Windows operating system are contained in the distribution). Compiling the C code is most easily accomplished by issuing the **mdpmexall** command. If you have never created a MEX file before MATLAB will search for a C compiler on your computer. MATLAB comes supplied with the LCC compiler but your computer may also have others available.[2] A menu will appear that requests that you choose one of the available compilers. Once you have chosen a compiler `mdpmexall` will proceed to compile all of the C files in the toolbox.

Although it is possible to run all of the toolbox procedures without creating MEX files they will tend to run more slowly and be more likely to encounter memory limitations. In some

[2] Apparently 64 bit versions of MATLAB are not supplied with a C compiler. Furthermore, the C code used in the toolbox has not been tested in a 64 bit environment so it should be compiled in 32 bit mode using the **-compatibleArrayDims** MEX option (you might need to edit `mdpmexall` to do this.)

cases the performance enhancements of the C code lead to run times that are hundreds of times faster and use as much as 4 times less memory. It is therefore highly recommended that these be created.

4 Getting Started

There are a number of common tasks that are typically required in specifying, solving and analyzing the results of a Markov decision problem. It is highly recommended that your MATLAB code follow each of these steps in clearly labelled blocks. This will facilitate troubleshooting and communication. The following list includes the common tasks

1. define problem parameters (constant values, vectors, matrices, functions, etc.)
2. create matrices of problem variables (states and actions)
3. define reward vector
4. define transition probability matrix
5. call solver
6. prepare and display results

To solve simple problems MDPSOLVE needs only 4 pieces of information, an n^x-row matrix specifying the values of the state and action variables, an n^x-element reward vector R, the discount rate δ and an $n^s \times n^x$ transition matrix P. To illustrate consider the problem of managing an invasive species or a pest infestation. The state variable takes on $n^s = 3$ values representing (1) no infestation (2) moderate infestation and (3) heavy infestation. There are $n^a = 2$ actions the manager can take: (1) do nothing and (2) apply a treatment. This implies that there are $n^x = 3 \times 2 = 6$ state/action combinations.

The matrix X lists the state/action combinations. Each column is a variable and in this problem there is a single state variable taking on values between 1 and 3 and a single action variable taking on values 1 and 2. The complete set of state/action combinations can be collected into the following 6×2 matrix with the action in column 1 and the state in column 2

$$X = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 2 & 1 \\ 2 & 2 \\ 2 & 3 \end{bmatrix}$$

Other orderings are possible (for example we could put the state in the first column or we could keep the values of the states together rather than the actions). The way that the variables are ordered has two features that will make it easy to keep track of stuff. First, the rows are

sorted lexicographically so the first column changes most slowly. MDPSOLVE has a variety of methods for generating matrices of variables and it always conforms to the lexicographic ordering. Second, the values of each action are kept together. This is useful here because it is convenient to define rewards and transition probabilities separately for each action.

The objective in this case is to minimize the damages from the pest plus any treatment costs incurred. The damages due to the pest are calculated to be 0, 5 and 20 for the 3 levels of infestation and the treatment cost is 10. Thus the reward vector (noting that rewards are negative costs) is

$$R = \begin{bmatrix} 0 \\ -5 \\ -20 \\ -10 \\ -15 \\ -30 \end{bmatrix}$$

Notice that the rows of R conform to the rows of X. The discount factor is taken to be 0.95; in general it is a number on $(0, 1]$.

The state dynamics are given by the transition probability matrix P^1 if no action is taken:

$$P^1 = \begin{bmatrix} 0.65 & 0.15 & 0.05 \\ 0.25 & 0.40 & 0.20 \\ 0.10 & 0.45 & 0.75 \end{bmatrix}$$

and the transition probability matrix P^2 if a treatment is applied:

$$P^2 = \begin{bmatrix} 0.85 & 0.45 & 0.35 \\ 0.15 & 0.50 & 0.50 \\ 0 & 0.05 & 0.15 \end{bmatrix}$$

To be well defined the probability matrices must have all non-negative elements that sum, over each column, to 1.

To solve this problem using MDPSOLVE one creates a script file which can be run at the MATLAB command line (see the file `pests_simple` in the `mdpdemos` directory). The first step is to define the parameter values (the data) for the problem.

```
% parameters
D = [0; 5; 20];  % damage costs
C = 10;          % spraying cost
delta = 0.95;    % discount factor


% state/action combinations
% columns: 1) value of action  2) value of state
X = [1 1;
     1 2;
```

```
      1 3;
      2 1;
      2 2;
      2 3];


% rewards
R = -[D D+C];


% probabilities with no action
P1=[0.65  0.15 0.05;
    0.25  0.40 0.20;
    0.10  0.45 0.75];
% probabilities with spraying
P2=[0.85 0.45 0.35;
    0.15 0.50 0.50;
    0    0.05 0.15];
P=[P1 P2];
```

To pass this information to `mdpsolve` it must be collected into a structure variable, which is a variable with information placed in named fields. The following code creates such a structure variable and passes it to the basic solver procedure `mdpsolve`:

```
% set up model structure
clear model
model.R = R;
model.d = delta;
model.P = P;
% call solution procedure
results = mdpsolve(model);
```

It should be pointed out that no time horizon for this problem has been specified; by default `mdpsolve` assumes that the problem has an infinite horizon. MDPSOLVE returns a structure variable with a number of different fields. If all goes well it will contain fields for the value function and for the optimal strategy (if the procedures encountered problems these are stored in the `errors` and `warnings` fields; more on that later). `results.v` stores the the optimal value function which is an $n^s \times 1$ vector of the values of the sum of current and discounted expected future rewards, conditioned on taking the optimal actions and evaluated for each of the n^s values of the current state. The second, `results.Ixopt`, is an $n^s \times 1$ vector $\mathcal{I}^{x*}$ indicating the rows of X associated with the optimal strategy. Under certain conditions the field `results.Aopt` is also provided. This contains the rows of X associated with $\mathcal{I}^{x*}$.

There are a variety of ways to display results. For a simple model like this simply listing the states and the associated optimal actions and value function is sufficient.

```
disp('Simple Pest Control Problem')
disp('State, Optimal Control and Value')
disp([results.Aopt(:,[2 1]) results.v])
```

This displays a 3×3 table of values. Column 1 is the value of the state, column 2 is the optimal action to take in each state and column 3 is the value associated with each state (the expression `X(:,[2 1])` reverses the columns of `Aopt` so the state is listed first).

If MDPSOLVE is installed correctly you should be able to type `pests_simple` at the MATLAB command line to run the script file. Doing so will cause a summary report and the following results to be displayed to your screen:

```
Simple Pest Control Problem
State, Optimal Control and Value
 1   1  -150.94
 2   2  -168.38
 3   2  -186.79
```

These results indicate that treatment ($A = 2$) should be applied when the infestation level is either moderate ($S = 2$) or heavy ($S = 3$). They also indicate the damages that can be expected. For example, even if there is no infestation now, so current damages are 0, it is expected that damages will occur in the future and hence the discounted expected future damages and treatment costs total to 150.94.

5 The General Framework

Discrete dynamic programming problems always involves 2 types of variables, states and actions and 2 functions of these variables, rewards and transition probabilities. In general there are n^s values that the state variable can take on and, for the ith state value, there are n_i^a values for the action. There are thus $n^x = \sum_{i=1}^{n^s} n_i^a$ state/action combinations.

The reward function can be defined as an n^x-vector that represents the current reward for each of the n^x state/action combinations. The transition probability can be thought of as an $n^s \times n^x$ matrix where the (i, j)th element represents the probability that the ith state will occur next period given that the jth state/action combination occurs in the current period. This matrix must consist of non-negative values that sum to 1 over each column.[3]

In solving a dynamic programming model it is necessary to know which of the n^x state/action combinations are associated with each of the n^s values of the current state. Suppose that we define an n^x-vector $\mathcal{I}^x$ with element j representing the state associated with the jth state/action combination. The problem then is to solve the Bellman equation for all n^s value of k:

$$V_k(t) = \max_{j:\mathcal{I}_j^x=k} \left[R_j + \delta \sum_{i=1}^{n^s} P_{ij} V_i(t+1) \right]$$

The Bellman equation provides a means of solving for $V(t)$ given $V(t+1)$. In finite horizon problems we take as given some terminal value $V(T+1)$ and use the Bellman equation to work backwards through time. If the time horizon is infinite, however, and the discount factor is strictly less than 1, the value is not a function of time and the Bellman equation becomes

$$V_k = \max_{j:\mathcal{I}_j^x=k} \left[R_j + \delta \sum_{i=1}^{n^s} P_{ij} V_i \right]$$

`mdpsolve` includes two methods for solving discounted infinite horizon problems; these are discussed in Section 10.1.

5.1 State/Action Combinations

Many treatments of dynamic programming require that there be the same number of actions for each state. For many problems this can lead to situations in which many of the state/action combinations are infeasible and result in memory limitations and/or excessive computing times. In order to accommodate situations in which the number of actions per state is not constant, MDPSOLVE defines problems in terms of the states and the state/action combinations. In many situations this is actually more natural and simpler to conceptualize: R is an n^x-vector with R_i representing the reward obtained when the current state/action combination is i and P is an $n^s \times n^x$ matrix with P_{ij} representing the probability that state j will occur in the next period when the current state/action combination is i.

[3]For technical reasons it is computationally more efficient in MATLAB to define the transition probabilities in column stochastic form with the columns summing to 1. The alternative row stochastic form is perhaps more common in mathematical treatments; for most purposes `mdpsolve` accepts them in either form.

Allowing P to be defined as a simple matrix rather than as a set of stacked matrices or as a 3-dimensional array (current state, current action, future state) simplifies the exposition. It also enables us to take advantage of matrix features of a language like MATLAB, which allows 3-dimensional arrays but limits what can be done with them (they cannot be sparse, many arithmetic operations are not defined for them, etc.).

To accommodate situations in which the number of actions (n^a) is the same for every state, the reward function can be defined as an $n^s \times n^a$ matrix and the transition probabilities as n^a matrices, each of size $n^s \times n^s$, stacked horizontally next to each other. This was the case for the simple example of Section 4.

In more complicated situations, however, it is possible that not all actions are feasible for every state. As an example of how this can arise consider a situation in which the state represents the level of some finite resource and the action is the amount of the resource used in the current period. Clearly the action cannot have a value greater than the value of the state. For example, suppose the values of the state are 0,1, 2, 3 or 4 and the values of the action are 0, 1 and 2 (it being assumed that it is infeasible to use more than 2 units of the resource in any period).

In this example there are 5 values of the state and 12 possible state/action combinations, which are listed in the following table:

j	$\mathcal{I}^x$	S	A
1	1	0	0
2	2	1	0
3	2	1	1
4	3	2	0
5	3	2	1
6	3	2	2
7	4	3	0
8	4	3	1
9	4	3	2
10	5	4	0
11	5	4	1
12	5	4	2

The table was generated using the following MATLAB code. In the first line the utility `rectgrid` (see Section 6.2.1) is used to generate all possible combinations of the two variables. The second line then eliminates the infeasible state/action combinations. The third line defines the two index vectors and combines these with the variable pairs.

```
X  = rectgrid((0:4)',(0:2)');
X=X(X(:,1)>=X(:,2),:);
tab = [(1:length(X))' X(:,1)+1 X];
```

In this example there is only 1 action for state 1 (state value 0), 2 actions for state 2 (state value 1) and 3 actions for each of states 3, 4 and 5 (state values 2, 3 and 4). More generally, if

there are n possible values of the resource level and $p \leq n$ possible utilization values, there are $p(p-1)/2$ infeasible state/action combinations if all np combinations were defined.

To facilitate this situation so infeasible state/action combinations need not be defined, an n^x-vector of index values `Ix` can be defined with each element equal to an integer between 1 and n^s. This index vector associates each of the n^x state/action combinations with one of the n^s values of the state. In the simple example in Section 4 in which all actions are feasible in all states the index can be defined by

```
Ix=repmat(1:ns,1,na)';
```

With $n^s = 3$, $n^a = 2$ and $n^x = 6$, this is the vector $[1\ 2\ 3\ 1\ 2\ 3]'$. In the more complicated resource utilization example, the `Ix` vector is given in second column of the table. The use of the `Ix` vector is discussed more fully in Section 6.4, where the utility `getindex` is described.

It should be pointed out that another way to rule out infeasible state/action combinations is to define them but require that the reward for these combinations be set to $-\infty$. Although this can simplify the exposition (see, for example, the treatment in Chapter 7 of Miranda and Fackler) it suffers from the disadvantage that the infeasible combinations are processed, resulting in greater use of memory and slower processing speeds. Additionally, it requires that fictitious transition probabilities be defined for these combinations (fictitious both because the probability is zero that these combinations occur and because they can never be chosen as optimal given the infinitely negative value of the associated reward). Although the transition probabilities can be arbitrarily defined for infeasible state/action combinations requiring them can lead to confusion. If you prefer this approach however it will work with `mdpsolve` but having the alternative allows one to define the problem in a computationally more efficient way that may also, to some users, seem more natural.

6 Defining Variable Values

Although not strictly necessary it is generally a good idea to define an n^x row matrix X with columns representing the various state and action variables and with each row representing a specific value of those variables. Checking to make sure that the variables are defined the way that you desire can save a lot of headaches in the long-run. If n^x is very big it is often possible to scale down the problem to a more manageable size to check it and then scale back up.

For example suppose that there are 2 state variables and 1 action variable. Suppose state variable 1 has 101 values ranging from 0 to 100, state variable 2 has 51 values ranging from 0 to 1 and the action variable has 21 values ranging from -1 to 1. This means that $n^s = 101*51 = 5151$ and $n^x = n^s * 21 = 108171$. Thus X is a 108171×3 matrix. If instead we set $n^s_1 = 5$, $n^s_2 = 4$ and $n^a = 3$ we get $n^s = 20$ and $n^x = 60$, which is far more manageable.

It is highly recommended that the parameters that define these grids be assigned variable names and that the variables be used in the remaining code. For example:

```
mins1=0; maxs1=100; ns1=5;
mins2=0; maxs2=1;   ns2=4;
mina=-1; maxa=1;    na=3;
s1=linspace(mins1,maxs1,ns1)';
```

```
s2=linspace(mins2,maxs2,ns2)';
a =linspace(mina, maxa, na)';
X=rectgrid(s1,s2,a);
```

Once it is determined that the variables are defined as you want them to be it is easy to change n_1^s, n_2^s or n^a.

It can also be very useful to keep a consistent way of ordering the values of the variables. In MDPSOLVE the ordering principle used is lexicographic, which is the way that dictionaries or telephone books are organized. We have already used the utility procedure `rectgrid` which creates rectangular or Cartesian grids. In the simple pest example we could use `X=rectgrid([1;2],[1;2;3]);` to create

$$X = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 2 & 1 \\ 2 & 2 \\ 2 & 3 \end{bmatrix}$$

This has the values of the action in column 1 and the values of the state in column 2.

If instead we used `X=rectgrid([1;2;3],[1;2]);` we would get

$$X = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 2 & 1 \\ 2 & 2 \\ 3 & 1 \\ 3 & 2 \end{bmatrix}$$

Not only is the first variable now the state but the ordering is different. The first approach organized the values in 2 blocks of 3 where the blocks represented different actions. Here there are 3 blocks of 2 where the blocks represent different values of the state.

In general put a variable first if you want to define things in terms of block with the variable having the same value. In many applications this will mean putting the action variables first and then the states as this will create blocks for which the action is constant over the block.

6.1 Indices versus Values

It is important to distinguish between indices and values. With n^s states, the index $\mathcal{I}^x$ is an n^x vector containing integers between 1 and n^s that uniquely identify each row of X with a particular state. The value of the state could be anything one wants. For example, suppose that the state represents the available amount of some resource and is represented by the values 0, 1, 3 and 6 (notice that these values need not be evenly spaced). The indices for these values are 1, 2, 3 and 4 (there actually is no requirement that the indices should have the same ordering as the values but it helps to avoid confusion).

This distinction can be especially confusing when the values are integers between 0 and N. In this case there are $N+1$ states and state values and so the indices take on integer values between 1 and $N+1$. C programmers might object that this is due to MATLAB's use of 1-based (rather than 0-based) indexing; a valid observation but only relevant if you want to program in C or some other 0-based language.

In the previous examples the state values are always single numbers but this need not be the case. We might, for example, have one state variable that takes on values 0, 1 and 2 and a second one that takes on values 0 and 1. The state values in this case are pairs of numbers. The following table demonstrates how indices and values can be related in this case.

	values	
i	S_1	S_2
1	0	0
2	0	1
3	1	0
4	1	1
5	2	0
6	2	1

6.2 Defining State and Action Grids

When the state values represent multiple variables it is useful to be able to easily generate all feasible values. Although this can happen in a wide variety of ways, two situations arise often enough that it is worth while having functions to perform this task. The first defines a rectangular (or Cartesian) grid in which a discrete set of points is defined for each of a collection of variables and a grid is formed from all possible combinations of points. The second applies to a collection of variables defined on a simplex, i.e., on a region that has non-negative values that collectively sum to a single number.

6.2.1 Rectangular Grids

Suppose that there are three variables that take on values in the sets $\{0, 1\}$, $\{1, 2, 3\}$, $\{-1, 1\}$, respectively. A complete rectangular grid is the set of triples

0	1	-1
0	1	1
0	2	-1
0	2	1
0	3	-1
0	3	1
1	1	-1
1	1	1
1	2	-1
1	2	1
1	3	-1
1	3	1

Although the order of these triples is arbitrary, it is useful to have a method for ordering them that you stick to. Notice here that the last variable varies the fastest and the first variable varies the slowest. This is the ordering that is obtained using the utility procedure `rectgrid`. It is called by passing to it the set of grid points for each variable:

```
S=rectgrid([0;1],[1;2;3],[-1;1]);
```

This creates the 12×3 matrix of points showed in the table above. A useful feature of this ordering is that it has the lexicographic property if the original vectors are each ordered from low to high. A lexicographic ordering is the way words are ordered in a dictionary or names in a phone book: the list is sorted first by the first variable, then by the second and so forth. Having a simple consistent ordering makes it much easier to keep track of the values in complex problems and the lexicographic ordering is always maintained in MDPSOLVE.

In this example variable 2 was defined with evenly spaced points. This need not be the case. A regular grid could have been defined in the same way had the values of variable 2 been 1, 2 and 4. The only difference is that interpolating on evenly spaced grids is slightly faster and so it is useful to provide this information when doing interpolation (for a discussion of interpolation procedures, see Section 9.2).

If you need to find the index number associated with a given value in the grid one can use `rectindex`. For example, suppose we use `s={[0;1],[1;2;3],[-1;1]}; S=rectgrid(s);` to generate the values given above. Notice that the lower case s variable is a three element cell array with the vectors that provide the grid points for each of the three variables. In general rectangular grids will be specified using cell arrays containing column vectors. We can use `rectindex([0 3 1],s)` to determine that the value $[1 \;\; 2 \;\; 0]$ is put in the 7th row of the matrix of values.

6.2.2 Grids on a Simplex

An alternative to rectangular grids arises when a set of state variables must contain non-negative numbers that collectively sum to a fixed number C. A simplex is an appropriate representation of a state in (at least) three situations. First, when there are N elements (sites, people, species) each of which can be in one of q categories. For example, there are N sites each of which can be uninfested, moderately infested or severely infested. The state could be described as the

number of sites in each category. Thus there are 3 state variables and the value of the state must sum to N. This can be called a category count model because the state represents the counts in each category.

A second situation is when the state variables represent the proportion of something that is in each category. The categories might represent, for example, the proportion of a population or a site that is in each of q different age or stage classes. Although each variable can take on a continuum of values between 0 and 1, we can discretize the variables using $p+1$ evenly spaced values on [0,1]. Each value is a q vector that sums to 1. When the variables represent the proportion of a geographic area in a given category this is equivalent to dividing the area into equal sized sites and treating the state as the number of sites in each category.

A final use is for representing beliefs in one of q mutually exclusive and exhaustive events. In this case the state variable is a q vector representing the degree of belief in each of the q states, with the state necessarily summing to 1.

In general suppose that there are q different variables each of which is non-negative and which collectively sum to C. Only $q-1$ state variables are needed to represent this situation because variable q is fully known once the values of the first $q-1$ variables are known. Furthermore, not all combinations of the $q-1$ variables are feasible because these variables must collectively sum to S or less.

When $q = 2$ the grid is fully represented by the set of possible values for the first variable and this is no different than a regular rectangular grid in 1-dimension. To illustrate the case when $q > 2$, suppose that $q = 3$ and each of the three variables can take on integer values between 0 and 3 (so $C = 3$). The complete set of the all three variables is

0	0	3
0	1	2
0	2	1
0	3	0
1	0	2
1	1	1
1	2	0
2	0	1
2	1	0
3	0	0

As with rectangular grids, the ordering used here is lexicographic, which is the way that dictionaries are sorted. This makes indexing the values (moving from the q values to a single index representing the place in the ordering of a given point) relatively easy. Grids defined on a simplex can be obtained using the utility procedure `simplexgrid` which has the calling syntax

```
S=simplexgrid(q,p,C);
```

Here q is the number of variables, p is the number of subintervals (so there are $p+1$ possible values for each variable) and C is the number that the variables must sum to (by default $C = p$; this would be appropriate for a category count situation, whereas $C = 1$ is the appropriate value for a belief state situation).

The procedure returns an $r \times q$ matrix of values where

$$r = \frac{(p+q-1)!}{p!(q-1)!} = \binom{p+q-1}{p} = \left(\!\!\binom{p}{q}\!\!\right)$$

(this is sometimes referred to as the multiset (p, q) coefficient). [4]

By default `simplexgrid` returns all q columns with each row summing to C. Use the syntax

```
S=simplexgrid(q,p,C,0)
```

to obtain a grid with $q-1$ columns and with each row summing to C or less.

If you need to find the index number associated with a given value in the grid one can use `simplexindex`. For example, suppose we use `S=simplexgrid(3,3,3)` to generate the values given above. We can use `simplexindex([1 2 0],3,3,3)` to determine that the value $[1\ \ 2\ \ 0]$ is put in the 7th row of the matrix of values.

6.3 Associating States and State/action Combinations

In general X is defined as an n^x-row matrix and some way of indicating which state value is associated with any specific value of X is required. One way to accomplish this is to actually specify the matrix X and also define a vector `svars` that lists which columns are state variables. If these are included as fields in the `model` structure variable `mdpsolve` can determine which states are associated with which actions (it will assume that the states are lists in lexicographic order).

As an example, in the pests example setting `model.X=X;` and `model.svars=2;` will allow `mdpsolve` to determine that $S = 1$ in the 1st and 4th place, $S = 2$ in the 2nd and 5th place and $S = 3$ in the 3rd and 6th place.

Actually what `mdpsolve` does is calls a utility `getI` to make this determination. You can save it the trouble by setting the `Ix` field in the model variable yourself. On way to obtain the I^x index is to use `[S,Ix]=getI(X,svars);` and then set `model.Ix=Ix;`. For some large problems there may be alternative ways to get the index values that are more efficient. For example, if the state variables are defined as elements of either a rectangular or simplex grid one can use `rectindex` or `simplexindex`.

A third alternative is available when there are the same number (n^a) of actions for each state (so $n^x = n^s n^a$). In this case the reward function can be defined as an $n^s \times n^a$ matrix. This assumes that the actions are first in the ordering of X, i.e., that all of the values of action i are in a contiguous block of n^s values. An example if given in the next section.

If X is defined using `X=rectgrid(A,S);`, where A contains n^a values and S contains n^s values, the $\mathcal{I}^x$ index can be obtained using

[4] When $C = p$ all of the values are non-negative integers. Although MATLAB allows variables to be defined using integer data types, which take up less memory than double precision floating point numbers, there is less support for these data types and they can be confusing to use. For this reason `simplexgrid` returns grids as double precision numbers. If this results in memory problems `simplexgrid` provides an option to define the grid using the least memory possible. Such a grid however should be used with caution, especially when performing arithmetic operations. For example, an integer times a floating point number is returned as an integer, meaning that significant information may be lost.

```
Ix=(1:ns)'*ones(1,na); Ix=Ix(:);
```

This orders X into n^a blocks each with n^s values. On the other hand if X is defined using `X=rectgrid(S,A);`, the $\mathcal{I}^x$ index can be obtained using

```
Ix=ones(na,1)*(1:ns); Ix=Ix(:);
```

This orders X into n^s blocks, each with n^a elements.

For example if S contains the integers from 0 to 3 and A is either 0 or 1 then the first ordering leads to

j	I^x	A	S
1	1	0	0
2	2	0	1
3	3	0	2
4	4	0	3
5	1	1	0
6	2	1	1
7	3	1	2
8	4	1	3

whereas the second ordering leads to

j	I^x	S	A
1	1	0	0
2	1	0	1
3	2	1	0
4	2	1	1
5	3	2	0
6	3	2	1
7	4	3	0
8	4	3	1

(the script file `Ixdemo` demonstrates this).

6.4 Different Numbers of Actions per State

In simple models the number of possible actions is unvarying across states. For this reason it is possible to define the rewards to be an $n \times m/n$ matrix. In more complicated situations, however, it may be better to allow a different number of actions per state. This can make a significant difference in the size of a problem when the state and/or action space is large.

As an example, consider a situation in which the are N sites to be managed and each site can be in one of two categories (call these 0 and 1). The state S is the number of sites in category 1, so there are $N+1$ state values ranging from 0 to N. Suppose that the action is the number of category 1 sites that receive some treatment. Thus the action is restricted to values ranging from 0 to S.

It is certainly possible to define such a problem allowing the action to take on any of $N+1$ possible values and constrain the action by making the reward for impossible state/action combinations equal $-\infty$. State/action combinations with infinitely negative rewards will never be

optimal and hence will not affect the solution. This results in $(N+1)^2$ possible state/action combinations, however, which can grow large as N increases.

If, instead, only feasible state/action combinations are considered, there are only $(N+1)(N+2)/2 \approx (N+1)^2/2$, thus eliminating nearly half of the state/action combinations. If processed efficiently, this can result in a great savings of time and memory usage.

Let n^x be the total number of feasible state/action combinations. Such a framework can be implemented by defining R to be a n^x vector, P to be an $n^s \times n^x$ matrix (or an $n^x \times n^s$ matrix if defined in row stochastic form) and defining an index vector `Ix` with n^x elements. `Ix` defines which state is associated with each of the n^x state/action combinations (so `Ix` contains integers between 1 and n^s).

If the `Ix` field is not defined it is assumed that $n^a = n^x/n^s$ is an integer value, that there are n^a possible actions for each state and that (implicitly)

```
Ix=repmat(1:ns,1,na)';
```

In general it does not matter whether `Ix` or R are defined as matrices or vectors so long as they have n^x elements and the elements are ordered correctly (MATLAB orders matrices so that rows increment first; if in doubt convert a matrix `M` to a vector using `M(:)`).

To illustrate this I've modified the simple pest example from Section 4 by defining `Ix` to be

```
[1  2  3  1  2  3]'
```

This is available in the script file `pests_Ix`.

Another advantage to this approach arises when actions are continuous but have been discretized. Having variable numbers of actions per state allows considerable flexibility in defining the fineness of the grid associated with different state values. This can result in the benefits of a finer grid mesh where it is needed (i.e., for states in which the problem is especially sensitive the the action) without making the problem grow to an unmanageable size required if all states have the same action grid.

This section is finished off with a somewhat complicated example. Suppose there are 3 sites that each can be in either of 2 categories and to which either of 2 treatments can be applied. This means there are 4 possible category/treatment combinations for each site. The state in such a model is the number of the 3 sites in each of the two categories and the value of the state is a 2-vector in which each of the two elements is a number between 0 and 3 with the proviso that the two numbers sum to 3. Similarly, the value of the state/action combinations is a 4-vector, with each element between 0 and 3 and with the 4 elements summing to 3. The state/action combinations represent the number of sites in each category/treatment combination.

In this example there are 4 values of the state and 20 possible state/action combinations, which are listed in the following table:

		state		state/action combination			
state index	$\mathcal{I}^x$	category 1	category 2	category 1 treatment 1	category 2 treatment 1	category 1 treatment 2	category 2 treatment 2
1	1	0	3	0	0	0	3
2	2	1	2	0	0	1	2
3	3	2	1	0	0	2	1
4	4	3	0	0	0	3	0
5	1	0	3	0	1	0	2
6	2	1	2	0	1	1	1
7	3	2	1	0	1	2	0
8	1	0	3	0	2	0	1
9	2	1	2	0	2	1	0
10	1	0	3	0	3	0	0
11	2	1	2	1	0	0	2
12	3	2	1	1	0	1	1
13	4	3	0	1	0	2	0
14	2	1	2	1	1	0	1
15	3	2	1	1	1	1	0
16	2	1	2	1	2	0	0
17	3	2	1	2	0	0	1
18	4	3	0	2	0	1	0
19	3	2	1	2	1	0	0
20	4	3	0	3	0	0	0

The table can be generated using the following code

```
X  = simplexgrid(4,3,3,1);
S   = double(X)*[1 0;0 1;1 0;0 1];
tab = [(1:length(S))' S(:,1)+1 S X];
```

The ordering this imposes comes from the lexicographic ordering of the state/action combinations used by the function `simplexgrid`, which is discussed in more detail in Section 6.2.2. An important point to note about this example is that there are 4 actions each for states 1 and 4 (which have values $[0\ 3]$ and $[3\ 0]$) and 6 actions each for states 2 and 3 (which have values $[1\ 2]$ and $[2\ 1]$).

Admittedly this example is a bit complex, even though the idea behind it is relatively simple. Fortunately `mdpsolve` has a set of tools for handling this type of problem, a so-called category count model (see Section 12.3). The important point here it that having a way to eliminate infeasible sites significantly reduces the problem size.

To see the magnitude of the reduction possible, consider that the problem can be defined with a single state variable that takes on 4 values (this can be interpreted as the number of sites in category 1) and three action variables that each can take on 4 values (these are the number of category 1 sites receiving treatment 1, the number of category 2 sites receiving treatment 1 and the number of category 1 sites receiving treatment 2). For both states and actions, there is an

implicit omitted variable which is equal to N minus the sum of the other variables (interpreted as the number of category 2 sites and the number of category 2 sites receiving treatment 2).

Defined in this way there are $4^3 = 64$ values of the actions is all possible state/action combinations are considered. This would require $m = 256$ (64 times the 4 state values). The infeasible combinations are those for which the actions sum to more that 3 and ones for which the sum of actions 1 and 3 does not equal the value of the state. This process can be carrying out using the following code;

```
S=rectgrid(repmat({(0:3)'},1,4));
S(sum(S(:,2:4),2)>3,:)=[];
S(S(:,2)+S(:,3)~=S(:,1),:)=[];
size(S)
```

The first line produces the 256 values (see Section 6.2.1 for a discussion of the `rectgrid` procedure), the second eliminates ones in which columns 2-4 (interpreted as the actions) sum to more than 3 and the third line eliminates ones in which actions 1 and 3 don't sum to the value as the the state in column 1. When these combinations are eliminated only 20 state/action combinations remain, as shown in the table above. This represents a huge reduction in the problem size.

7 Defining Rewards

For each of the n^x values of the state/action combinations there is an associated reward value. For small problems it may be easiest to simply list the n^x values. For example one could use `R=[1;2;3;4;5]';` to define the rewards for a problem with $n^x = 5$. A slight elaboration of this idea arises when the reward is the sum of a reward the depends on the state and another that depends on the action. For example, in the simple pest infestation example in which $n^s = 3$ and $n^a = 2$ we could define the reward using `R=[-D -D-C]`, where D is a 3×1 vector of damages associated with the states and C is a scalar cost that is incurred if the site is treated.

In general, however, it is often easiest to use the values in X to generate the reward. For example, if the first column of X contains the action and the second column the state we could use

```
R = -D(X(:,2)) - C*(X(:,1)==1);
```

To see how this works note that the value of the state, in column 2 of X is either 1, 2 or 3 and therefore can be used to index the elements of D. In the second term the expression `X(:,1)==2` returns a vector of 0s and 1s with a 1 whenever X_2 equals 2.

If a reward is a complicated expression it is a good idea to break it up into smaller pieces and/or to write a separate function to compute it. In MATLAB you can define so-called anonymous functions as single lines of code or you can define a function in an m-file. For example

```
cost1 = @(S,A) D(S) + C*(A==2);
R = -cost1(X(:,2),X(:,1));
```

defines the anonymous function `cost` as a function of the state and the action. This is somewhat easier to read than defining it in terms of X.

Alternatively one could create the following function and save it as the file `cost2.m`:

```
function R=cost2(S,A,D,C)
  R = D(S) + C*(A==2);
```

There is an important difference between these two approaches. Using the m-file approach requires that one pass the parameters D and C to the the function, whereas these are stored with the anonymous function. This means that if you change the parameter values, you also need to redefine `cost1` because the function uses the previously stored values. It also means that D and C must be created prior to creating the anonymous function.

One could also combine these two approaches:

```
cost3 = @(S,A) cost2(S,A,D,C);
```

This creates an anonymous function that calls the m-file. Like `cost1` it uses the existing values of D and C. This is actually an attractive alternative because it means that the m-file version can be written once and then many anonymous functions can be created using alternative parameter values. This can be useful for performing sensitivity analysis and for use in models with multiple stages.

8 Alternative Ways to Define Transition Matrices

One of the most important limitations on the size of dynamic programming problems involves defining the transition matrices. This is especially problematic when the number (n^x) of state/action combinations gets large. This section deals with a number of features of `mdpsolve` that address this problem.

The basic operation of `mdpsolve` uses the transition probability matrix P, where P_{ij} represents the probability that state i will occur next period given that state/action combination j holds in the current period (this assumes the P is in column stochastic form, so the columns sum to 1). When possible it is best to compute this matrix once and store it in memory so it can be accessed quickly and efficiently. A number of the methods described in this section concern ways to reduce the memory needed to store the transition matrix.

One of the most important ways of reducing both memory requirements and processing time is to use sparse matrix methods. Sparse matrices store and process only the non-zero elements of a matrix. If there are many non-zero elements in a probability matrix, as is often the case, the potential exists for substantial efficiencies. MATLAB incorporates sparse matrix methods fairly seamlessly with non-sparse (dense) storage methods making them very easy to use once a few commands are learned. The main command needed is `sparse` which creates a sparse matrix using vectors of row and column indices and a vector containing the non-zero values. An important point to note about sparse matrices is that different ways of populating them have dramatically different memory usage and execution time implications (see

`http://www.mathworks.com/help/techdoc/math/f6-8856.html#f6-33214`

for a discussion).

MDPSOLVE can process probability matrices defined in either dense or sparse form. If the sparse form is used for moderate to large problems, it can be far more efficient to define and process the transition matrix when it is in column stochastic form so rows are associated with future states and columns with state/action combinations. Many of the extended features of MDPSOLVE create transition matrices in column stochastic form (this is true for `g2P`, `f2P`, `amdp`, `pomdp`, and `catcountP`). For the most part `mdpsolve` can detect whether a problem is defined in row or column stochastic form and takes the appropriate action (it never hurts, however, to set the `model.colstoch` field to either 0 or 1 so appropriate checks can be conducted).

The main use of P in dynamic programming algorithms is to compute the expected value function in the next period given the current state/action combination: $EV(S^+)|X]$. For a given value of P this can be expressed as $P^\top V$ where V is an n^s vector containing next period's value function for each state value (if P is in row stochastic form this would be simply PV). It is possible that a problem is simply too large for the transition matrix to be held in memory or one wants to use a method that does not require that P be formed explicitly. In this case one can define the transition information as a function that accepts an n^s vector V^+ and returns an n^a vector $E[V^+|X]$. Setting `model.EV=true;` enables this feature.

There are also often alternative ways to define a MDP when one or more of the state variables is actually continuous. For continuous state variables the state transitions can be defined either in terms of a transition density function $f(S^+|X)$ or in terms of a state transition equa-

tion, which maps the current state/action combination and a random variable into next period's state variable: $S^+ = g(X, e)$. Ways to model problems with continuous state variables are discussed in Section 9.

8.1 Eliminating Redundant Probabilities

For some problems alternative state/action combinations can result in the same transition probabilities. Suppose, for example, that the state is the level of a non-renewable resource and the action is the amount of the resource utilized this period. If the probabilities associated with given states depend only on the post action level of the resource then any state/action combinations with equal values of $S - A$ have the same probabilities.

To make this explicit, suppose that there are 4 possible levels of the resource, 0 through 3, and the action can take any value between 0 and 3 so long as it is not greater than the state. This results in $n^x = 10$ feasible state/action combinations as shown in the following table. Of the 10 combinations, however, there are only 4 values of $S - A$ and hence one need only define P with 4 rows and 4 columns. Which of these 4 values a given state/action combination is associated with is given as `Iexpand` in the third column. There are also a different number of actions for each value of the state so `Ix` must be defined; it is shown in the second column of the table.

state/action index	$\mathcal{I}^x$	I^{expand}	S	A	$S - A$
1	1	1	0	0	0
2	2	2	1	0	1
3	3	3	2	0	2
4	4	4	3	0	3
5	2	1	1	1	0
6	3	2	2	1	1
7	4	3	3	1	2
8	3	1	2	2	0
9	4	2	3	2	1
10	4	1	3	3	0

It is, of course, not necessary that one identify all (or even any) identical rows of P but this example makes clear that the size of the P matrix can be reduced considerably when this is done.

To implement this method, the `model.Iexpand` field should be set to the `Iexpand` index vector. Although it may not be necessary it is a good practice to set the `model.colstoch` field to inform the solver whether future states are associated with columns (`colstoch=1`) or with rows (`colstoch=0`) of P. It is also useful to set the `model.ns` field to the number of state values. When you use the Iexpand feature it is possible (though unlikely) that `mdpsolve` cannot determine whether the transition matrices are in row or column stochastic form. In this case an error message will result; setting the `model.colstoch` field will eliminate this problem.

An example of the use of the `Iexpand` feature is provided in `mine_Iexpand`, which implements the resource extraction problem using a reward function of the form

$$\left(p - \frac{A}{1+S}\right) A$$

where p is the price per unit obtained in selling the extracted resource. The code uses both `Iexpand` and `Ix`; the former allows the P matrix to be defined as an $n \times n$ sparse identity matrix and the latter eliminates infeasible state/action combinations in which the action (the amount extracted) is greater than the state (the remaining amount of the resource), which results in $n(n+1)/2$ state/action combinations rather than n^2.

8.2 The EV Option

As discussed above it may be desirable to have `mdpsolve` use your own function for computing $EV(S^+)|X]$. To implement this set `model.EV=true` and define P as a function that accepts an n^s vector V^+ and returns an n^a vector $E[V^+|X]$.

Here is a somewhat mindless example but one that illustrates the point. Suppose that $n^s = 3$ and $n^x = 6$ (this should be familiar by now. Further suppose that you have defined a 3×6 transition matrix and named it `Pmat`. Define

```
model.P = @(V) Pmat'*V;
```

Of course this does the same thing as setting `model.P=Pmat` but it does illustrate the principle.

For a more interesting us of the the `EV` feature suppose that the state transition is given by the deterministic function

$$S^+ = \frac{c_1 S}{1 + c2 * S}(1 - A)$$

This is an example of a continuous state variable for which the transitions do not necessarily fall on a grid point. To address this the following code could be used:

```
X           = rectgrid(s,a);
recruitment = @(S,A) c1*S./(1+c2*S).*(1-A);
Splus       = recruitment(X(:,1),X(:,2));
EV          = @(V) pchip(s,V,Splus);
```

Here `pchip` is a MATLAB procedure that interpolates using a so-called shape preserving technique. This interpolation method cannot be expressed in the form $P^\top V$ because the interpolation method is not linear in V. Using such an interpolation method would not be possible without the `EV` feature. See `fishEV` for a demonstration of this idea.

In general, the use of the `EV` feature will slow down the solution but it can also aid in obtaining solutions when P is too large to fit in memory.

8.3 Merging Groups of State Variables

For many problems with multiple state variables, it can be challenging to keep all of the indices straight. `mdpsolve` has a procedure to help with this process. Suppose that you have multiple groups of variables that can be defined either independently of one another or with one group conditional on another. The state for the full problem is composed of all possible combinations of the group states and the associated probabilities can be computed using tensor (Kronecker) products.

An example will illustrate. Suppose the first subgroup has 3 state values

$$S_1 = \begin{bmatrix} 0 & 2 \\ 1 & 1 \\ 2 & 0 \end{bmatrix}$$

with transition probability matrix P_1 and the second subgroup has 2 state values

$$S_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

with transition probability matrix P_2. The combined states are

$$S = \begin{bmatrix} 0 & 2 & 0 \\ 0 & 2 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 2 & 0 & 0 \\ 2 & 0 & 1 \end{bmatrix}$$

and $P = P_1 \otimes P_2$. Notice that the states in both subgroups are ordered lexicographically and that the combined states preserve this ordering.

The utility `mergestates` can be used to facilitate this process. It accepts a set of cell arrays each containing a matrix of state values and a probability matrix and returns the merged values. The syntax is:

```
[S,P] = mergestates(C1,C2,...,options);
```

where the `Ci` are defined as cell arrays containing the S and P matrices, i.e., {`Si,Pi`}. The `options` input is a structure variable with (currently) the single allowable field `colstoch` which is set to 0 if the probability matrices are in row stochastic form (the default is 1). Note that all of the matrices must be in either column or row stochastic form.

When there are state/action combinations associated with the probability matrices and an `Ix` index vector associating state/action combinations with states these may also be passed to `mergestates` and updated. To do this simply enclose an n^x-row matrix of state/action variable values and an n^x-vector of state indices in the associated cell array. The syntax becomes:[5]

```
[S,P,X,Ix] = mergestates(C1,C2,...);
```

where the `Ci` are defined as either {`Si,Pi`} or {`Si,Pi,Xi,Ixi`}. If only the states are to be merged then `Ci` can be set to `Si` or to {`Si`}.

[5] If any of the `Pi` are omitted or empty, P is returned as an empty matrix. If `Xi` and `Ixi` are omitted or empty, `Ixi` is assumed to be `(1:size(Si,1))'`.

For example, suppose in the first subgroup that in either state 1 or 3 an action variable can take on values 0 or 1 and in state 2 the action must have value 0. The associated action and index vectors are given in the following table

A	$\mathcal{I}^x$
0	1
1	1
0	2
0	3
1	3

When merged with the second group (which is not associated with any actions) the combined action and index becomes

A	$\mathcal{I}^x$
0	1
0	2
1	1
1	2
0	3
0	4
0	5
0	6
1	5
1	6

The assumption is that the feasibility of the actions does not depend on the value of the state in the second group. If this is not true it is a simple matter to delete rows of the action and index matrices and the associated columns of P (or rows if P is in row stochastic form).

The example just discussed can be demonstrated using the following code, which is also available in the script file `mergestatesdemo`:

```
S1=[0 2;1 1;2 0]; P1=rand(5,3); P1=dvecxmat(1./sum(P1,2),P1);
A1=[0;1;0;0;1]; Ix1=[1;1;2;3;3];
S2=[0;1]; P2=rand(2,2); P2=dvecxmat(1./sum(P2,2),P2);
[S,P,A,Ix]=mergestates({S1,P1,A1,Ix1},{S2,P2});
disp(SS), disp([AA II])
```

The `mergestates` utility can be used sequentially and will produce the same results as if it is called once. For example

```
[S,P,A,Ix] = mergestates(C1,C2);
[S,P,A,Ix] = mergestates({S,P,A,Ix},C3);
```

produces the same results as

```
[S,P,A,Ix] = mergestates(C1,C2,C3);
```

When the groups of states are not probabilistically independent the problem of merging collections of states is more difficult. In this case `mergestates` can still be used but with the limitation that only 2 subgroups can be passed to it (more groups can be merged but they must be merged sequentially). For conditional merging either `P1` or `P2` should be defined as a function handle for a function that accepts a single input. The function should take a single row of the `X` matrix for the other group and return the transition probability conditional on that value of `X`. More specifically, if group 2 is conditional on group 1, `P2` should return a matrix with

$$P_{2ij}(X_1) = Prob(S_2^+ = s_{2j} | X_1, X_2 = x_{2i})$$

An example should help clarify the situation. Suppose that there are two groups of state variables. The second group consists of the pairs (S_{21}, S_{22}) with values (0,2), (1,1) and (2,0). Furthermore in states 1 and 3 the actions 0 or 1 can be taken and in state 2 only the action 0 can be taken. The first state can take on the values 0 or 1 and there is no associated action.

Suppose further that the transition probabilities associated with the first group are given by

$$P_1 = \begin{bmatrix} 1 & 0 \\ 0.47 & 0.53 \end{bmatrix}$$

The probabilities associated with the second group, however, are conditional on the value of the first group:

$$P_1(S_1{=}0) = \begin{bmatrix} 0.44 & 0.24 & 0.32 \\ 0.07 & 0.73 & 0.19 \\ 0.44 & 0.24 & 0.32 \\ 0.35 & 0.34 & 0.31 \\ 0.00 & 0.41 & 0.59 \end{bmatrix}$$

$$P_2(S_1{=}1) = \begin{bmatrix} 0.36 & 0.43 & 0.21 \\ 0.21 & 0.37 & 0.42 \\ 0.46 & 0.13 & 0.42 \\ 0.37 & 0.21 & 0.42 \\ 0.57 & 0.14 & 0.29 \end{bmatrix}$$

If the first group is placed first in the merged ordering we would obtain the following result.

				0	0	0	1	1	1	S_1
				0	1	2	0	1	2	S_{21}
S_1	S_{21}	S_{22}	A	2	1	0	2	1	0	S_{22}
0	0	2	0	0.44	0.24	0.32	0	0	0	
0	0	2	1	0.07	0.73	0.19	0	0	0	
0	1	1	0	0.44	0.24	0.32	0	0	0	
0	2	0	0	0.35	0.34	0.31	0	0	0	
0	2	0	1	0.00	0.41	0.59	0	0	0	
1	0	2	0	0.17	0.20	0.10	0.19	0.23	0.11	
1	0	2	1	0.10	0.17	0.20	0.11	0.20	0.23	
1	1	1	0	0.21	0.06	0.19	0.24	0.07	0.22	
1	2	0	0	0.17	0.10	0.20	0.20	0.11	0.22	
1	2	0	1	0.27	0.07	0.14	0.30	0.08	0.15	

Here there are 4 5×3 blocks. The upper left hand block is $P_1(S_2 = 0)$. The lower left hand block is $P_1(S_2 = 1)$ times 0.47 and the lower right block is $P_1(S_2 = 1)$ times 0.53.

If the second group is placed first in the merged ordering we would obtain the following result.

				0	0	1	1	2	2	S_{21}
				2	2	1	1	0	0	S_{22}
S_{21}	S_{22}	S_1	A	0	1	0	1	0	1	S_1
0	2	0	0	0.44	0	0.24	0	0.32	0	
0	2	1	0	0.17	0.19	0.20	0.23	0.10	0.11	
0	2	0	1	0.07	0	0.73	0	0.19	0	
0	2	1	1	0.10	0.11	0.17	0.20	0.20	0.23	
1	1	0	0	0.44	0	0.24	0	0.32	0	
1	1	1	0	0.21	0.24	0.06	0.07	0.19	0.22	
2	0	0	0	0.35	0	0.34	0	0.31	0	
2	0	1	0	0.17	0.20	0.10	0.11	0.20	0.22	
2	0	0	1	0.00	0	0.41	0	0.59	0	
2	0	1	1	0.27	0.30	0.07	0.08	0.14	0.15	

The order is not so obvious now but this is simply a rearrangement of the previous matrix.

9 State Transitions for Continuous Variables

The basic solution approach to solving dynamic optimization problems used by MDPSOLVE assumes that the states and actions can be represented by discrete finite sets of values. In practice, of course, variables can really take on a continuum of values. It is therefore essential that one have some method or methods to adequately represent problems with continuous variables by approximate models with discrete variables.

There are two natural ways to represent transition dynamics for continuous states. The first is perhaps the most common; it consists of expressing the value of the next period state in terms of the current state/action combination and, if stochastic, a random shock:

$$S^+ = g(X, e)$$

where the shock has some specified probability distribution. The other method represents the transition with a conditional probability density function $f(S^+|X)$.

9.1 State Transition Matrices vs. Functions

The basic program `mdpsolve` defines state transitions using transition probability matrices with elements P_{ij} that define the probability that the system moves to state i given that the current state/action combination is j:

$$P_{ij} = Prob(S^+ = s_i | X = x_j)$$

It is often more natural, however, to define the transition dynamics in terms of a function g of the form

$$S^+ = g(S, A, e)$$

where A is an action and e is a random variable (or random vector); e is sometimes referred to as a shock or as environmental variation or noise.

There are two essential tasks involved in the use of state transition functions. First, a set of discrete state values must be defined. How to define grids using MDPSOLVE was discussed in Section 6.2. The second task is to define probability matrices on this discrete set. For this task the procedure `g2P` can be used.

Before this procedure can be used, the state transition function must be defined and to do so requires that the environmental shocks and the associated probability distribution be defined. It is assumed that there are a discrete number (n^e) of values that e can take on and each of these is associated with a probability weight w. For w to be valid all n^e of its values must be no less than 0 and must sum to 1: $w_k \geq 0$ and $\sum_{k=1}^{n^e} w_k = 1$. These should be defined with the rows of both e and w corresponding to the individual realizations of the random variable and hence both e and w must have n^e rows.

In the simplest case there is a single random variable so both e and w are defined as n^e vectors. For example, suppose that e can take on the values 0, 1 and 2, with probabilities 0.3, 0.5 and 0.2. These are defined using

```
e=[0;1;2];
w=[0.3;0.5;0.2];
```

A more involved example is one with 2 random variables; in this case e can have 2 columns. For example suppose the first random variable (e_1) is defined as in the previous paragraph and that the second (e_2) is independent of e_1 and takes on the values 0 or 1 with probabilities 0.4 and 0.6. There are 6 possible realizations of these two random variables and hence $n^e = 6$. Thus

$$e = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \\ 2 & 0 \\ 2 & 1 \end{bmatrix}$$

and

$$w = \begin{bmatrix} 0.3 \cdot 0.4 \\ 0.3 \cdot 0.6 \\ 0.5 \cdot 0.4 \\ 0.5 \cdot 0.6 \\ 0.2 \cdot 0.4 \\ 0.2 \cdot 0.6 \end{bmatrix} = \begin{bmatrix} 0.12 \\ 0.18 \\ 0.20 \\ 0.30 \\ 0.08 \\ 0.12 \end{bmatrix}$$

In this example the joint probability of getting any specific realization of e_1 and e_2 is equal to the product of the marginal distributions. This is not true unless the two random variables are independent (actually this is the definition of independence).

The simplest way to obtain the joint random terms and the associated probabilities is to use `mergestates`, a precedure described in Section 8.3. For example, the above model can be obtained using

```
[e,w]=mergestates({[0;1;2],[0.3;0.5;0.2]},{[0;1],[0.4;0.6]});
```

An important issue concerns converting continuous random variables into a discrete set. There are a number of ways to do this, including the use of quadrature nodes and weights (see, e.g., [4], Chapter 5, for discussion). MDPSOLVE is distributed with procedures for obtaining Gaussian quadrature nodes and weights for the multivariate normal, Gamma and Beta distributions (`qnwnorm`, `qnwgamma` and `qnwbeta`, respectively). As an example, suppose we are interested in discretizing a normal distribution with mean 2 and variance 3 using a 5 point discretization. This can be accomplished using

```
[e,w]=qnwnorm(5,2,3);
```

The procedure `g2P` converts a functional representation of a transition to a transition probability matrix form. The key issue in making this conversion is that the function g does not generally produce values of S^+ that fall exactly a grid points. This means that some form of interpolation and, possibly, extrapolation is needed.

An interpolation method can be viewed as a way of approximating one function with another. Suppose we want to approximate $f(x)$ using only information about the function values at a discrete set of points, i.e., the input data is a set of pairs (x_i, f_i) where $f_i = f(x_i)$. Many interpolation methods can be viewed as defining a set of weights on each grid point and taking a weighted sum of the values at each of the grid points:

$$\hat{f}(x) = \sum_{i=1}^{n} w_i(x) f_i$$

If these weights are all non-negative and sum to 1, they can be viewed as probability weights.

The simplest form of interpolation is so-called nearest neighbor interpolation which puts a weight of 1 on the nearest grid point and 0 on all of the others. This method, however, generally does not produce accurate results unless the grid mesh is very fine.

A more accurate method that produces reasonable results is to use linear interpolation. If there is only one state variable linear interpolation puts non-zero weights on the points just below and just above. Thus to interpolate the point 1.4 on the grid $(0, 1, 3)$ would put weights 0.8 on grid point 2 and 0.2 on grid point 3: $0.8 = (3-1.4)/(3-1)$. Notice that the grid points need not be evenly spaced.

When the state is multidimensional linear interpolation requires that the grid be regular. Regular grids are ones that are composed of every combination of points for each variable. Such a grid can be defined by specifying a set of vectors, one for each variable, that list the grid points for that variable. Note that a regular grid is not required to be evenly spaced.

There are many cases in which regular grids are not appropriate for state variables. The most common of these is when a set of state variables must sum to a particular number. For example, suppose that state variables 1 and 2 represent the number of sites in one of two categories, where there are N sites. The sum of these two states must equal N.

States of this type are defined on a geometric form known as a simplex rather than on a rectangular region and require a different approach to interpolation (this approach is implemented for some of the model extensions discussed in Section 12, including adaptive management, partial observability and category count models). A limitation of such grids is that each of these variables must have the same number of grid points or the grid will not be regular enough for regular triangulation to be used. The procedures implemented in MDPSOLVE require that the grid points are evenly spaced.

One issue that arises when discretizing continuous state problems is the problem of extrapolation. It is sometimes the case that S^+ is outside of the discrete grid for certain values of S, A and e. More accurately, S^+ may lie outside the convex hull defined by the discrete values of S.[6] When this happens, some of the probability weights will be outside of the $[0, 1]$ interval.

This problem does not generally happen when a state variable is defined on a simplex or on a bounded rectangular grid. It does arise, however, if the state variable is not naturally bounded but a finite grid is used to represent the state space. For example, a state variable can take on any non-negative value must be truncated at some point and it is possible that S^+ may fall outside of the truncation point.

[6]The convex hull of a set of points (x_i) is that set of values that can be expressed as convex combinations of the x_i, i.e., the set of points for which there exists a vector w with $w_i \geq 0$ and $\sum_i w_1 = 1$ such that $x = \sum_i w_i x_i$.

The easiest way to handle this issue is to simply not worry about it. Negative probabilities can still be used by `mdpsolve` and useful solutions may be found (though a warning message is issued). It is, however, a good idea to perform some sensitivity analysis to see if results of interest are affected by extrapolation. For example, one might change the truncation points and see what affect this has on the solution.

If it is important to have valid probabilities `g2P` employs a simple way to "clean up" the extrapolation "problem." When one or more of the P_{ij} are negative we can obtain a closely related probability matrix by setting

$$\hat{P}_{ij} = \frac{\max\left(0, \sum_k w_k \lambda_{ij}(e_k)\right)}{\sum_i \max\left(0, \sum_k w_k \lambda_{ij}(e_k)\right)} = \frac{\max(0, P_{ij})}{\sum_i \max(0, P_{ij})}$$

This can be obtained by passing an options structure to `g2P` with the `options.noextrap` field set to 1. A potential problem with this approach is that it is possible in some situations to create an absorbing state at the boundaries of the state grid. This could cause misleading solutions to be returned. In most cases the use of this option has little effect on a well specified problem other than to eliminate annoying warning messages.

In some cases it is useful to allow the probabilities (w) associated with the noise terms (e) to be state and/or action dependent. This case can be handled by `g2P` by making w a $n^x \times n^e$ matrix. With this feature it is possible to have `g2P` return the transition matrix for a discrete state model (this is not a useful feature but it illustrates the way that state dependent probabilities can be handled). Specifically, if we define S to be

```
S=(1:n)';
```

and

```
g = @(X,e) e;
```

and then call

```
Pf=g2P(g,s,X,S,P);
```

The returned value `Pf` from `g2P` will be P or its transpose (`g2p` always returns transition matrices in column stochastic form). This is demonstrated in the script file `pests_g2P`.

9.2 Interpolation

When the state transitions are defined with a state transition function rather than a matrix, the problem of interpolation of non-grid points arises. `mdpsolve` has two interpolation methods available, one for rectangular grids and the other for grids defined on a simplex. Currently only rectangular interpolation is implemented in `g2P` but interpolation on a simplex is available independently, as described below.

Both of these types of grids suffer from curse of dimensionality problems. In the future it hoped that the state space will be able to be defined over non-regular collections of points. This will (potentially) enable problems of higher dimensionality to be solved but requires that different interpolation methods be defined.

9.2.1 Rectangular Grids

The function `rectbas` (which stands for rectangular basis) can be used to interpolate using this grid. Its calling syntax is

```
B=rectbas(x,s,evenspacing,noextrap);
```

Here x is a matrix of n^x evaluation points and s is a cell array containing the vectors of values for each variable. The procedure creates an $n^s \times n^x$ interpolation matrix where n^s is the total number of grid points. When all of the grid points are evenly spaced it is more efficient to use the syntax

```
B=rectbas(x,s,1);
```

(the third input argument is 1 for evenly spaced points and 0 otherwise, with 0 as the default value). By default `rectbas` will extrapolate if an evaluation is outside of the convex hull defined by the grid points. This results in interpolation weights that can be either negative or greater than 1. The fourth input argument can be set to 1 to force all of the weights to be non-negative and sum to 1. It does this by setting all of the negative weights to 0 and dividing all of the positive weights by their sum.

Suppose v is a function that maps values on a rectangular region into scalar values and written so $v(S)$ is an $n^s \times 1$ vector of values of the function at each of the n^s values of the grid. Interpolated values for the points in x can be obtained using

```
v(x)'≈ v(S)'*B(x)
```

9.2.2 Grids on a Simplex

Interpolation can be accomplished using `simplexbas`, which had the calling syntax

```
B=simplexbas(x,q,p,C)
```

This creates an interpolation matrix. Suppose v is a function that maps values on a simplex into scalar values and written so $v(S)$ is an $n^s \times 1$ vector of values of the function at each of the n^s values of the grid. Interpolated values for the points in x can be obtained using

```
v(x)'≈ v(S)'*B(x)
```

To accomplish interpolation on a simplex `simplexbas` uses a modified version of what is called Freudenthal triangulation. This is described in some detail in the papers [3] and [6]. There is further discussion of this algorithm in Section 15.1.

One restriction on simplex grids is that the number of subintervals for each of the q variables forming the collection must be equal to a common value of p. This is not really an issue in category count models, where the values of the state are naturally defined to be integers between 0 and C. In belief state situations, however, the grid provides a way to discretize a continuous space and may place too many points in areas of the space where a less dense set would be adequate and not enough in areas where a more dense set would be useful. Although it is possible (in principle) to define irregular grids (both on a simplex or a rectangular regions) these techniques are not implemented in MDPSOLVE (yet).

9.3 An Alternative Discretization Method

In the previous sections it is assumed that the transition is expressed in terms of a function that maps the current state/action X and a noise term e into the future state S^+. An alternative is to express the transition in terms of the conditional density function $f(S^+|S, A)$.

Recall that what we want to accomplish is to get estimate of

$$\int f(S^+|X)V(S^+)dS^+$$

This is the domain of numerical quadrature and there is a large literature on ways to accurately approximate integrals. Without going into this too deeply, suppose that we have an quadrature method that gives us

$$\int \pi(S)f(S)dS \approx \sum_i w_i f(s_i)$$

Then

$$\int f(s|x_j)V(s)ds = \int \pi(s)\frac{f(s|x_j)}{\pi(s)}V(s)ds \approx \sum_i \frac{w_i f(s_i|x_j)}{\pi(s_i)}V(s_i) = \sum_i P_{ij}V(s_i)$$

Thus one can define probabilities using

$$P_{ij} = \frac{w_i f(s_i|x_j)}{\pi(s_i)}$$

MDPSOLVE has a procedure, `f2P`, to create a transition matrix from a conditional density function. The basic syntax is

```
P=f2P(f,S,X,w);
```

where f is a function that computes the conditional density. It should accept the matrix of state values S and a single row of the state/action matrix X and return an n-vector of density values. The last argument, w is optional. This is a weight vector of the same length as S and is equal to $w_i/\pi(s_i)$ in the notation above.

To illustrate, consider the transition equation

$$S^+ = \mu + \alpha(S - \mu) + \sigma e$$

where $e \sim N(0, 1)$ is a standard normal noise term. This model is mean-reverting and the long-run distribution is $N(\mu, \sigma^2/(1 - \alpha^2))$. Rewriting this in terms of e

$$e = \frac{S^+ - \mu - \alpha(S - \mu)}{\sigma}$$

This implies that

$$f(S^+|S) \propto \exp(-0.5e^2)$$

It is unnecessary to determine the scaling constant that ensures the density integrates to 1 as the values of the discrete distribution will be scaled to sum to 1. This can be implemented using

```
n=21; a=-1; b=3;
mu=1; alpha=0.95; sigma=0.15;
S=linspace(a,b,n)';
f=@(S,X) exp(-0.5*((S-mu-alpha*(X-mu))/sigma).^2);
P=f2P(f,S,S);
```

The procedure `f2P` has two additional optional inputs. It can be called using

```
P=f2P(f,S,X,w);
```

where w is a weight vector of length n^s (the number of elements in S). These would be determined by the quadrature method the underlies the approximation. The procedure can also be called using

```
P=f2P(f,S,X,w,tol);
```

if `tol` is omitted or empty the transition matrix that is formed is dense, which could cause memory issues. If `tol` is positive then any value of P less than tol$/n^s$ will be set to 0 and the matrix will be sparse.

Even if the state transition is defined in terms of a function, it is often fairly easy to transform it into a condition probability density. Suppose that $S^+ = g(X, e)$ with the probability density function for e given by $f_e(e)$. We need to be able to solve for e in terms of S^+ and X to obtain something of the form $e = h(S^+, X)$. Then the conditional distribution of S^+ is

$$f_{S^+}(S^+|S) = f_e(h(S^+, S)) \left| \frac{\partial h(S^+, S)}{\partial S^+} \right|$$

(I know this breaks the no calculus rule but it would be difficult to get around it here.)

For simple, single variable problems the transformation is often very straightforward. A few are given here.

S^+		$f(S^+\|S)$
$g(X) + \sigma e$	$E[e] = 0$	$\frac{1}{\sigma} f_e\Big((S^+ - g(X))/\sigma\Big)$
$g(X)e$	$E[e] = 1$	$\frac{1}{g(X)} f_e\Big(S^+/g(X)\Big)$
$g(X)\exp(\sigma e)$	$E[\exp(\sigma e)] = 1$	$\frac{1}{\sigma S^+} f_e\Big(\ln(S^+/g(X))/\sigma\Big)$

10 Miscellaneous Features

10.1 Solution Methods

The solution method used to solve an MDP depends on the time horizon and the discount rate. If the time horizon if finite MDPSOLVE used backwards iteration of the Bellman equation given on p.11. If the time horizon is infinite and the discount factor is strictly less than 1, MDPSOLVE provides the options to use function iteration, policy iteration or modified policy iteration. If the horizon in infinite and the discount factor is 1, the problem is one of ergodic control, which is discussed further in Section 10.3.

Function iteration uses the Bellman equation to update the value function V, starting with an arbitrary guess of V. It can be shown that iterating on the Bellman equation will eventually cause V to converge to the solution in which V maps into itself (this is due to the fact that the iteration is a contraction mapping). The algorithm is superficially like the one used with a finite horizon but it differs in two ways. First, the solution does not depend on the starting value and, second, the iterations continue until a convergence criteria is met rather than for a fixed number of iterations. The convergence criteria is to stop when the maximum absolute change in the value function vector is less that $(1-\delta)\epsilon$, where δ is the discount factor and ϵ is a user controlled tolerance level. It can be shown that meeting the convergence criterion ensures that the value function is within ϵ of the true solution. The tolerance can be set by specifying the `options.tol` field; the default is 10^{-8}.

MDPSOLVE has an alternative convergence criterion that can be useful if the value function converges very slowly but the optimal action is found very quickly. MDPSOLVE will keep track of how many iterations have passed with no change in the conditional optimal action vector and will stop if this number exceeds some user defined number. This is set using the `options.nochangelim` field. The default is ∞; i.e., MDPSOLVE does not use this criteria unless specifically requested to do so. To use a different value set the `options.nochangelim` field equal to the desired value.[7]

An alternative to function iteration is policy iteration, sometimes known as Howard iteration. Policy iteration proceeds in the following way. First, the optimal action is determined for a guess of V. This provides the conditional maximizing action, which is used to form R^* and P^*. The value function is then updated by solving

$$(I - \delta P^*)V = R^*$$

Policy iteration requires a linear solve at each iteration, which can be numerically time consuming if n^s is large.[8] As a general rule when policy iteration is feasible it is preferred over

[7] This approach should be used with caution because the value function may converge very slowly, especially if the discount factor is near 1. This could cause the action to remain constant for many iterations even though it is suboptimal.

[8] If P is a dense matrix a linear solve requires $O(n^3)$ operations. If P is sparse, however, the number of operations can be far smaller and will depend on the number of non-zeros that P contains. If for a given state/action combination only a fixed number of future states can be reached the linear solve may actually require only $O(n^2)$ operations.

function iteration because it typically requires far fewer steps to achieve convergence. For this reason, policy iteration is the default algorithm for infinite horizon problems.

Like function iteration, policy iteration continues until a convergence criterion is met. In this case, however, one simply examines the change in the actions. If no change has occurred, the procedure has converged.

Both function and policy iteration will stop if a user controlled maximum number of iterations are exceeded. This can be controlled by setting the `options.maxit` field. The default is $250\ln(n^s)$ for function iteration and $20\ln(n^s)$ for policy iteration.

A third option is so-called modified policy iteration. Although it can be viewed as a modification of policy iteration and hence its name, it is perhaps more usefully viewed as a modification of function iteration. In function iteration there are really two steps. The first step is to determine the optimal action given the current "guess" of the value function, i.e., to solve

$$\max_A R^A + \delta P^A v$$

The second step is to update v using this policy. In modified policy iteration one instead finds the best A and then performs a number of updates using

$$v \leftarrow R^A + \delta P^A v$$

This approach addresses the fact that the best policy often remains the same through many function iterations and thus it may not be necessary to compute the value function for all n^x state/action combinations and perform a maximization operation. Instead simply iterate by computing the n^s values of v using the same policy. These iterations can be performed until some convergence tolerance is met or until some maximum number of sub-iterations is reached, at which point one starts over with a new function iteration.

This approach has generally been found to be nearly as fast (or faster) than policy iteration and does not require a linear solve (or indeed any need to explicitly form the transition matrix). The default approach taken by `mdpsolve` when function iteration is requested (by setting `options.algorithm='f'`) is actually to use modified policy iteration with a default maximum number of sub-iterations equal to 100. The maximum number of sub-iterations can be specified using `options.modpol`. Setting this to 0 produces true function iteration. For problems that are solved repeatedly experimenting with this option can result in large time savings.

10.2 Discounting

The model field `discount` is used to specify a discount factor. This factor measures the value of a reward obtained in the next period relative to the same reward obtained in the current period. The discount factor should generally be a number greater than 0 and less than or equal to 1. When it is 1, it means that rewards in all periods are given equal value. In general, the larger the value, the more weight is placed on future rewards. If the value of the discount factor were 0, the problem would be to pick the action that maximizes the current reward, with no regard to future rewards.

Although typically the discount factor is a single number, it can also be state dependent, in which case the `discount` field should be set to an n^s-vector.

If the model is defined with stages, different discount rates can apply to each stage (see Section 11). Like P and R, the discount field can be set to a cell array of stage relevant elements or can be set to a function. In the latter case the function should accept the time counter as a single argument and should return either a scalar or an n^s-vector.

Hyperbolic discounting could be implemented for finite horizon problems by defining the discount factor as a cell array of values, one for each period. There is currently a large and growing literature on the merits of hyperbolic versus exponential discounting but this is not the place to delve into the issues raised in this literature.

10.3 Average Reward (Ergodic) Control

When the discount factor is set to 1 so the future is not discounted, an infinite horizon problem is often not well posed because the value function may increase without limit. In this case the relationship

$$V = \max_A R(S,A) + P(S,A)V^+$$

is still valid but

$$V \neq \max_A R(S,A) + P(S,A)V$$

(as is often mistakenly stated in the literature). The value function itself does not necessarily converge.[9]

With no discounting of the future it is generally more useful to solve the problem of maximizing the average per period reward over an infinite horizon, a problem known as ergodic control. In an ergodic control we attempt to find the decision rule $A(S)$ that solves

$$v(S) = \max_{A(S)} \lim_{T\to\infty} \frac{1}{T+1} \sum_{t=0}^{T} E\Big[R\Big(S_t, A(S_t)\Big)\Big|S_0 = S\Big]$$

A sufficient condition for such a solution to exist for finite Markov chains is that the chain is recurrent, meaning that it is possible to attain any state starting at any other state. A less restrictive sufficient condition is that there is a single recurrence class, i.e., there is a unique set of states that, once entered, cannot be exited (recurrent chains are ones with a single recurrence class that includes all of the states).

In an ergodic control problem in which P^A has a single recurrence group, the value function does not depend on the state because the current state will have no impact on the long run

[9] If the second relationship held it would be possible to use a policy iteration approach and solve

$$(I - P(S,A))V = R(S,A)$$

The solution to this equation is not uniquely defined, however, as the fact that 1 is an eigenvalue for any probability matrix in turn implies that $I - P$ is singular.

behavior of the system and hence on the average value of the rewards obtained. Thus v is simply a number. In this case the long run average expected reward is equal to $\sum_{i=1}^{n} \pi_i(A)R_i(A)$ where $\pi_i(A)$ is the long run probability of visiting state i given policy rule A. For example, in the simple pest model it can be found using

```
x=[1 5 6]; R(x)*longrunP(P(:,x))
```

which results in -8.0172 (see `pests_ergodic`).

There are a number of ways to solve ergodic control problems but the simplest of them, known as the vanishing discount method, is to use a discount factor that is very close to 1. In the pest problem, if we use a discount factor of 0.999999, the value function is close to constant (the values differ in the 6th significant digit) and $(1-\delta)v$ is equal to -8.0172, which is equal to the value obtained using the long-run distribution. To see why note that, when δ is near 1,

$$E\left[\sum_{i=0}^{\infty} \delta^i R(x_t)\Big|S_0\right] \approx \sum_{i=0}^{\infty} \delta^i E[R(x)] = \frac{1}{1-\delta}E[R(x)]$$

(this result holds even if a suboptimal decision rule used).

Rather than actually setting `model.delta` MDPSOLVE allows you to set `options.vanish` to a value close to but less than 1. It will use this value as the discount factor if the discount factor specified by the model variable is 1. Before returning the results, however, it will adjust the value function to return an approximation of the average value function. As the value of `options.vanish` approaches 1 the value function should approach a constant value equal to the average expected reward.

An alternative is to simply use function iteration over a long enough time (recall that you cannot use policy iteration when the discount factor is 1). One criterion is to iterate until the control rule is unchanged for a specified number of iterations. This can be accomplished by setting `options.nochangelim` to the desired value; it equals ∞ by default.

Both of these approaches can be problematic, however. With δ very close to 1, it may be difficult to solve the linear system $(I-\delta P)v = R$ because $(I-\delta P)$ is nearly singular. If function iteration is used, it can take a long time to converge (rates of convergence in function iteration depend on δ). On the other hard it is difficult to know a priori when to stop iterating if $\delta = 1$ and there is no guarantee that the solution has converged.

An alternative approach uses the Bellman like equation known as the Average Reward Optimality Equation (AROE)

$$v + h = \max_A R^A + P^A h$$

where v is a scalar equal to the long-run average reward and h is a vector. The AROE is the basis for an alternative algorithm, known as the relative value approach, which is the default in MDPSOLVE for problems with a discount factor equal to 1. Notice that any scalar can be added to both sides of the AROE and therefore one can set one of the values of h arbitrary to 0. Setting `options.relval` equal to a positive integer between 1 and n^s activates the relative value method.

It is important to note that the value function returned is the expected total reward relative to value in the state specified by `options.relval`. The value of the average reward is a scalar

value that is returned in the `results.AR` field. It is this number that should be compared to the value obtained using the vanishing discount approach. To simplify these comparisons this field is set to the mean of the adjusted value function when using the vanishing discount approach.[10]

Although the average expected reward is useful for many problems, there are some problems in which the relative value function is more useful. An example of this arises when there is an absorbing state that is entered with probability 1. Suppose the absorbing state is state 1 and we set `options.relval=1`. The relative value function, returned as `results.v`, can be interpreted as the total reward before absorption.

The script file `pests_ergodic` demonstrates the use of the various features for handling non-discounted problems. It is important to point out that the theory that applies to non-discounted problems is considerably more complicated that for the discounted case and whether a solution even exists depends on the nature of the transition matrix. There are situations in which the relative value approach does not yield a solution. It is generally a good idea to try alternative algorithm and verify that they give consistent results.

There is yet another approach that one can take to obtain solutions to ergodic problems. The ergodic strategy can be defined in terms of the probability that a given state/action occurs, w_j. We would like to maximize the expected rewards:

$$\max_{w} \sum_{j} w_j R_j$$

subject to the w_j satisfy $w_j \geq 0$, $\sum_j w_j = 1$ and

$$\sum_{j} P_{ij} w_j = \sum_{j:\mathcal{I}_j^x = i} w_j$$

The last condition ensures that the probabilities are consistent with the long run behavior of the state process.

Unlike the standard DP framework this approach allows strategies to be random. There is no reason why two values of j associated with the same state (i.e., for which $\mathcal{I}_j^x = \mathcal{I}_{j'}^x$) cannot both be non-zero. The main advantage of this framework, however, is that additional constraints of the form $\sum_j w_j f(X_j) \lessgtr b$ can be imposed. In words, such a constraint ensures that the the expected value of some function of the system variables is less than, equal to or greater than some target value. This approach uses linear programming to obtain a solution. Look for this feature in a future version of MDPSOLVE; the current version does include a linear programming solver (`lpsolve`) if you want to try it.

10.4 Starting Values and Terminal Conditions

In finite horizon problems a terminal condition must be specified. The terminal condition, $V(T+1)$, is an n^s-vector with the ith element representing the value of being in state i at time

[10] It is actually possible to use both methods simultaneously and to use the relative value approach with discounted problems.

$T+1$. The default value is that $V(T+1) = 0$, which implies that whatever happens after time T does not figure into the problem being addressed. Likewise, if $V(T+1)$ is equal to any constant it means that the state of the world after time T does not figure into the problem being addressed. This might indeed be the case but it certainly deserves some thought. To specify a given value of $V(T+1)$ set the `model.vterm` field to an n^s-vector of choice.

In infinite horizon problems there is not pre-specification of $V(T+1)$; in fact for discounted infinite horizon problems $V(T+1)$ is the same as for any other time period. It is possible and even desirable to specify a starting vector when using either function or policy iteration. This can be done by setting the `options.v` field that is passed to `mdpsolve`. Although the initial value of V seems a look like the terminal value $V(T+1)$, it is not actually part of the model specification and hence it is specified in the `options` structure rather than the `model` structure.

The default value for the initial value is an n^s-vector of 0s. It is possible, however, to speed up the solution procedure by picking an initial guess that is closer to the solution. This might be obtained from introspection or from the solution to a closely related problem. Good starting values are especially useful if function iteration is used.

10.5 Convergence Checks

When a problem has a finite time horizon the solution is found using standard backwards recursion. When the problem is an infinite horizon problem, however, we are essentially attempting to find a fixed point and have choices concerning both the algorithm used and the convergence criteria.

Two methods for solving infinite horizon problems are implemented in MDPSOLVE, function iteration and policy (or Howard) iteration. The first uses the same backwards iteration operator that is used for finite horizon problems. Iterations continue until convergence criteria are met or the maximum allowable number of iterations has been reached. The main convergence criteria makes use of the fact that the backwards iteration operation is a contraction mapping satisfying (see Judd, p.413)

$$||V^* - V^k||_\infty \leq \frac{1}{1-\delta}||V^{k+1} - V^k||_\infty$$

This tells us three things: (1) that convergence to the solution V^* is linear, i.e., that the error is reduced at a constant rate, (2) that convergence is faster when δ is smaller and (3) that to achieve an error in the value function of less than ϵ, we can stop iterations when $||V^{k+1} - V^k||_\infty \leq (1-\delta)\epsilon$. If V is very large or very small one might need to adjust the value of ϵ.

If policy iteration is used in a problem with a discrete set of actions, the convergence criterion is very simple. Check whether any of the actions change from one iteration to the next; if not, the optimal value function and action has been found.

When only the decision rule and not the value function is needed, another criterion that can be used with function iteration is to count the number of iterations since the action last changed and stop if this number exceeds a preset limit (this is specified by setting the `options.nochangelim` field). The idea here is that the value function iteration can continue long after the optimal action is determined and, if the value function is not needed, these iterations are not necessary.

The problem here is that it is not necessarily the case that convergence has been achieved. One could check this by taking one policy iteration step using the computed value function as a starting point for the iterations (by setting the `options.v` field).

These relatively simple convergence criteria apply only to models without stages. Models with stages need to be handled somewhat more carefully. See Section 11 for further discussion.

11 Models with Stages

So far we have considered only models with a single $n^s \times n^x$ transition matrix. Many models are more intuitively designed and efficiently processed in terms of a set of stages. For example, consider a situation in which a set of sites changes first by succession, in which sites in one successional type may, with some probability, move to the next successional type. In the next stage, which might represent a dry season, there is some probability of fire, which changes a site to an early successional type. The dynamic programming algorithm alternates between the two stages. More generally stages can be used to represent different seasons or different life stages.

Stages are specified by defining the `model.nstage` field to be greater than 1 and by defining one or more of the `model.discount`, `model.R` and `model.P` as cell arrays with `nstage` elements. In addition, the `model.Ix` or `model.Iexpand` fields can also be defined as cell arrays if they change definitions over the stages. For example, suppose that rewards, transitions or actions differ between summer and winter. One could define `model.nstage=2` and both `model.R` and `model.P` as 1×2 cell arrays.

It is possible to replicate a stage multiple times before going to the next stage information. Suppose, for example, that an insect population reproduces 3 times during a breeding season and then enters a winter dormant stage. This could be modeled using 2 stages with the first one replicated 3 times and the second replicated only once. This can be specified by setting `model.nrep=[3 1]`. In general `model.nrep` is a vector of length `nstage` composed of positive integers. Note that if `nstage` is 1 then `nrep` is ignored.

Strictly speaking, it is not necessary to specify `model.nstage` as it can be inferred from the size of the cell arrays used to define the data fields. Nonetheless it is a good idea in general to pass more model information rather than less as this allows more checks to be made to ensure that the model is defined as you intend it to be.

To see how a stage model can be used, consider the following example that is concerned with controlling an invasive pest (yes, pests again). The states are the degree of infestation (no, medium, high). The transition is broken into two stages, a growth stage and a die-back stage. Two actions are possible, no-treatment and treatment. To make this specific, suppose that in the growth phase the transition matrices are

$$G^n = \begin{bmatrix} 0.65 & 0 & 0 \\ 0.25 & 0.55 & 0 \\ 0.10 & 0.45 & 1 \end{bmatrix} \text{ and } G^t = \begin{bmatrix} 0.85 & 0 & 0 \\ 0.10 & 0.90 & 0 \\ 0.05 & 0.10 & 1 \end{bmatrix}$$

Here treatment slows the growth of the infestation. The transition probability matrices for the die-off stage are

$$D^n = \begin{bmatrix} 1 & 0.30 & 0.10 \\ 0 & 0.70 & 0.20 \\ 0 & 0 & 0.70 \end{bmatrix} \text{ and } D^t = \begin{bmatrix} 1 & 0.80 & 0.50 \\ 0 & 0.20 & 0.30 \\ 0 & 0 & 0.20 \end{bmatrix}$$

Damages are only measured before the growth phase, with the damage and treatment costs and discount factor as before (damage costs of 0, 5 and 20 for the 3 levels of infestation, treatment cost of 10 and discount factor of 0.95).

See the script file `pests_stages` for a complete demonstration of this model. Once the parameters (data) are defined the script is

```
model.R        = {[zeros(3,1) zeros(3,1)-C];[-D -D-C]};
model.d        = {1;delta};
model.P        = {[Gn Gt], [Dn Dt]};
model.T        = inf;
model.nstage   = 2;
model.nrep = [1 1];
results=mdpsolve(model,options);
```

It is important to note that the value function and optimal control returned by MDPSOLVE with staged models will have multiple dimensions. For example, v and $\mathcal{I}^{x*}$ that are returned in the staged pest example are both 2 element cell arrays, with each element being a 3 vector. More generally there will be n^{stage} elements in each where the ith element is an $n^s(i) \times n^{rep}(i)$ matrix. Element $v\{i\}(j,k)$ is the value of the value function when the state is the jth state in stage i, replication k. The script produces the following output

```
Staged Pest Control Problem
State, Optimal Control and Value
1   1   1  -176.64  -167.81
2   1   1  -196.92  -186.30
3   1   2  -209.91  -209.91
```

It is not necessary for there to be multiple actions in a given stage. Suppose, for example that one could only treat during the second stage (which is the optimal strategy anyway). The following model definition only allows a treatment option in the second stage.

```
model.R        = {zeros(3,1); [-D -D-C]};
model.discount = {1;delta};
model.P        = {Gn, [Dn Dt]};
model.T        = [2 inf];
```

Not surprisingly, this produces identical output as before because treatment was not optimal in the first stage.

Given that the optimal action only involves treatment at one of the stages it is possible to treat this model as a non-staged model. Using

```
model.R        = -[D D+C];
model.discount = delta;
model.P        = [Dn*Gn Dt*Gn];
model.T        = inf;
```

Note the order in which the stage transition matrices are multiplied to get the full cycle transition matrices; if the probability matrices were given in transposed form, the order should be reversed because $(AB)^\top = B^\top A^\top$). A useful way to remember this is that the columns of the first matrix are associated with current state, they should match the rows of the earlier transition matrix because of the future values of the earlier stage are the current values of the later stage. This model produces

```
Non-staged Pest Control Problem - Action taken after growth phase
State, Optimal Control and Value
1   1   -167.81
2   1   -186.30
3   2   -209.91
```

which is the optimal action and value for the second stage.

In principle any field in the model variable can be defined separately for any given stage. If a field only contains a single numeric value or a 1 element cell array, that single element will be applied to all of the stages. If a field is undefined, `mdpsolve` will attempt to find the appropriate value for this field based on the other information provided.

Algorithms and Convergence Issues

For finite horizon models stages are handled in the same fashion as models with only a single stage. For infinite horizon models function iteration and policy iteration are both implemented (modified policy iteration is not implemented for models with stages). To implement policy iteration it is necessary that the rewards and transition matrix for a full cycle be built up from the rewards and transition matrices for each stage. It is possible that memory limitations will be encountered in forming the transition matrix. In this case one should switch to function iteration.

For infinite horizon problems convergence is not quite so simple in models with stages. The reason for this is that the value function is different at each stage. Thus we do not want to check convergence with respect to the value function for the previous iteration but rather with respect to the value function obtained one complete cycle previously.

MDPSOLVE addresses this issue in function iteration by keeping track of the value function for every stage and tracking the maximal change over all stages and replications. Convergence is then checked once per complete cycle.

With policy iteration the convergence criteria is to check whether the actions change relative to the maximizing action found one complete cycle previously. Policy iteration is more difficult to implement and might result in memory issues. In particular, it requires that the optimal transition matrix over a whole cycle be computed and stored. If it works, however, it still tends to be result in faster processing unless n^s is very large, especially if the transition probabilities can be stored in a dense matrix format.

Rewards and Transitions Defined by Functions

It is possible to specify either or both the reward matrices (R) and transition matrices (P) as functions rather than as matrices. Although this can be done for models with only 1 stage, there is no compelling reason to do so. For problems with stages, however, defining them as functions allows the values to be computed as needed rather than stored. For simple problems this feature is not particularly useful but can be useful in avoiding memory limitations in models defined with stages.

To implement this feature one should pass a cell array of function handles in either the `P` or `R` fields of the model variable. These functions should have no inputs. `R` should return an n^x vector and `P` should return an $n^s \times n^x$ matrix if `model.colstoch=true` (or an $n^x \times n^s$ matrix if `model.colstoch=false`;).

To see how this can used, suppose that there are two stages and that matrices $S\{1\}$, $S\{2\}$, $X\{1\}$ and $X\{2\}$ have been defined, as have the conditional transition densities $f\{1\}$, and $f\{2\}$. Using

```
for i=1:2, model.P{i}=@()f2P(f{i},S{i},X{i}); end
```

creates function handles that call the procedure `f2P` (discussed in Section 9.3) passing in the appropriate information and returning the desired $n^s \times n^x$ matrix. See `pests_stages_func` for an example of how this feature can be utilized. It uses a different approach than that just described in that it creates the transition matrices once and stores them to disk. The utility `saveget` automates this process to some degree although it leaves behind `.MAT` files with funny names that should simply be erased after the program is run.

12 Extending the Basic Framework

A number of specialized model types can be formulated and solved using the MDPSOLVE. In each case these are formulated by modifying a initial model by passing transition and reward matrices to an auxiliary function and using the results from that function to create a new model which can be passed to `mdpsolve`.

Currently three such auxiliary functions are available, each of which defines new state variables on a simplex and sets up the transition matrices using an interpolation method designed for simplexes. The three specialized model types are

- Models with Model Uncertainty: `amdp`
- Models with Partial Observability: `pomdp`
- Category Count Models: `catcountP`

12.1 Adaptive Management with Model Uncertainty

It is often the case there is uncertainty about the transition model. One way to address this is to define a set of q alternative models and to expand the state space to include a set of q variables, the kth of which (B_k) represents the degree of belief one has in model k. The value of these variables is never negative and must sum to 1, i.e. they lie on a unit simplex.

Given a current belief state, the future belief state for model k is defined using Bayes Rule

$$B_k^+(i,j,B) = Prob(\text{model} = k | S^+ = i, X = j, B) = \frac{P_{ij}^k B_k}{\sum_{k=1}^q P_{ij}^k B_k}$$

The objective for an infinite horizon adaptive management problem is to find the decision rule $A(S,B)$ that solves

$$V(S,B) = \max_{A(S,B)} \sum_{t=0}^{\infty} E\left[\delta^t R\Big(S_t, A(S_t,B_t)\Big)\Big| S_0 = S, B_0 = B\right]$$

given the transition dynamics of S and B. For $0 < \delta < 1$ this problem satisfies the Bellman equation[11]

$$V(S,B) = \max_{A(S,B)} R\Big(S, A(S,B)\Big) + E\left[\delta V(S^+, B^+)|S,B\right]$$

To make this operational, the belief space for B is discretized using p subintervals for each belief variable. The updating rule does not guarantee that the updated belief falls on one of these grid points and therefore an interpolation rule must be used to define the transition probabilities for the belief states. Belief state variables are defined on a simplex with q non-negative values that sum to 1, resulting in

$$N = \frac{(p+q-1)!}{p!(q-1)!}$$

[11] When $\delta = 1$ this Bellman relationship does not hold because V does not converge.

feasible grid points (see Section 6.2.2 for a discussion of grids and interpolation on a simplex).

Any value in the belief space can be interpolated using the values at the q nearest grid points. This results in a "probability" matrix with exactly q non-zero elements per node, rather than the 2^{q-1} required with linear interpolation (when there are only two models ($q = 2$) the two interpolation methods are identical). This method of interpolation is implemented in the function `simplexbas`; the grid is initialized by calling `simplexgrid`.

Transition matrices as well as modified reward functions and state values can be obtained using the procedure `amdp`. This procedure accepts as inputs p (the number of belief intervals for each state value), P (a cell array containing q transition matrices), and R and returns updated versions of P and R. If also passed `S`, `X` and `Ix` the procedure updates these as well. The belief states are placed last in the list of states and the state values are ordered lexicographically. The resulting values can be placed in a model structure and passed to `mdpsolve`.

Passive Adaptive Management

The function `amdp` is used to obtain the so-called "actively" adaptive management solution. The passively managed version does not treat the beliefs as state variables that are updated. Instead, for each value of the belief set, the beliefs are treated as fixed so that no learning is anticipated and belief updating is not required to obtain the optimal control rule. Instead, obtaining a solution involves solving a set of problems of the same size as if the model was known with certainty.

The objective for an infinite horizon passive adaptive management problem is to find the decision rule $A(S, B)$ that solves

$$V(S,B) = \max_{A(S,B)} \sum_{t=0}^{\infty} E\left[\delta^t R\Big(S_t, A(S_t,B)\Big)\Big| S_0 = S, B\right]$$

given the transition dynamics of S and B (for simplicity it is assumed here that R does not depend on the dynamics of S). This problem satisfies the Bellman equation

$$V(S,B) = \max_{A(S,B)} R\Big(S, A(S,B)\Big) + E\left[\delta V(S^+, B)|S, B\right]$$

The difference between this and the active management is that, in obtaining the optimal solution $A(S, B)$, it is assumed that B remains the same for all time. Once the solution is obtained, one can apply this decision rule along with Bayes Rule to update B in implementing the management strategy. This implies that B does change but that this fact is ignored in obtaining the optimal decision rule.

MDPSOLVE also has a procedure `amdppassive` that solves passive adaptive management problems. It's syntax is

```
[v,a,B]=amdppassive(p,P,R,model,options);
```

p is the number of belief intervals for each model, P is a cell array of model specific transition probabilities, R is a cell array of model specific rewards. All of the other information about the model is contained in the structure variable `model`. If either P or R are passed as empty, the information concerning how they are specified is assumed to be non-model specific and

available in the appropriate field of the `model` variable. This procedure will edit the `model` variable appropriately and pass it and the `options` variable to `mdpsolve` repeatedly until the problem has been solved for every value of the belief states.

Algorithm Details

In active adaptive management we need to compute

$$E[V(S^+, B^+)|S, B, A]$$

The updated beliefs, B^+ are deterministic once S^+ is known (one simply applies Bayes Rule to obtain these), so

$$E[V(S^+, B^+)|S, B, A] = \sum_{i=1}^{n} Prob(S^+ = s_i|S, B, A)V(s_i, B^+)$$

Suppose that we have discretized the belief space into a set of N belief values b_u, where each b_u is a q-vector. The first part of this expression is simply the belief weighted sum of the model probabilities:

$$Prob(S^+ = s_i|S = s_j, B = b_u, A) = \sum_{k=1}^{q} [b_u]_k P_{ij}^k(A) = \hat{P}_{iju}(A)$$

To determine $V(s_j, B^+)$ we need to interpolate with respect to B^+ as it need not fall at a grid point. Suppose we write $V(s_i, b_v) = V_{jv}$ and we approximate $V(s_i, B^+)$ using

$$V(s_i, B^+) \approx \sum_v w_v(s_i, B^+)V_{jv} = \sum_v w_{ijuv}V_v$$

The interpolation weights are written with 4 subscripts because B^+ is a function of $S = s_i$, $S^+ = s_j$ and $B = b_u$. Putting this together we obtain

$$E[V(S^+, B^+)|X = x_i, B = b_u] = \sum_{i=1}^{n^s}\sum_{v=1}^{N}\left[\hat{P}_{iju}w_{ijuv}\right]V_{iv} = \sum_{i=1}^{n^s}\sum_{v=1}^{N}\tilde{P}_{ijuv}V_{iv}$$

When the approximation weights are nonnegative and sum to 1 (as is true of Freudenthal interpolation on a simplex) the $\tilde{P}_{ijuv}$ terms will also be nonnegative and sum to 1 over i and v. Thus the $n^sN \times n^sN$ matrix $\tilde{P}$ defines a transition probability matrix for the active adaptive management problem. Letting $\hat{R}_{ju} = R(x_j, b_u)$,[12] the Bellman equation for the active adaptive management problem is

$$V_{ju} = \max_{A(s_j,b_u)} \hat{R}_{ju}(A) + \delta \sum_{i=1}^{n^s}\sum_{v=1}^{N}\tilde{P}_{ijuv}V_{iv}$$

[12] If there are q model specific reward functions $\hat{R}_{ju}(A) = \sum_{k=1}^{q}[b_u]_k R_j^k$.

For the passive adaptive management problem the situation is simpler. Here the problem is to compute

$$E\left[V(S^+,B)|S,B,A\right] = \sum_{i=1}^{n} Prob(S^+ = s_i|X,B)V(s_i,B)$$

(it should be noted that this V is not necessarily the same V that solves the active adaptive management problem). The difference is that B^+ does not appear in the future value function and thus no interpolation is needed. Specifically

$$E[V(S^+,B)|X = x_j, B = b_u] = \sum_{i=1}^{n^s}\left[\sum_{k=1}^{q}[b_u]_k P_{ij}^u(A)\right] V_{ju} = \hat{P}_{iju}V_{iu}$$

The $n^s \times n^s$ matrix $\hat{P}_{iju}$ is a belief weighted sum of the individual model transition matrices. Notice that everything here is constant in u and hence passive adaptive management problems can be solved by treating separately each of the N belief points. The Bellman equation for the passive adaptive management problem is

$$V_{ju} = \max_{A(s_j,b_u)} \hat{R}_{ju}(A) + \delta \sum_{i=1}^{n^s} \hat{P}_{iju}V_{iu}$$

Example: Managing a Pest Infestation

Consider the problem of managing a pest infestation (this example is modified from a paper by [1]). A potentially infested site has $n = 3$ possible levels of infestation: (1) none, (2) medium and (3) high. The cost of the damages caused by the infestation are calculated to be 0, 10 and 20. A manager can do nothing or can treat the infestation at a cost of 20. The complete reward matrix is

$$R = \begin{bmatrix} 0 & -20 \\ -10 & -30 \\ -20 & -40 \end{bmatrix}$$

The discount factor is assumed to be 0.95.

If treated, the transition probability matrix is

$$P^t = \begin{bmatrix} 0.9 & 0.8 & 0.6 \\ 0.1 & 0.2 & 0.4 \\ 0.0 & 0.0 & 0.0 \end{bmatrix}$$

(Note that `amdp` requires that the probability matrices be in column stochastic form). When the site is left untreated there is uncertainty concerning how the pest spreads. Consider two models which differ in their untreated transition matrices:

$$P^1 = \begin{bmatrix} 0.9 & 0.6 & 0 \\ 0.1 & 0.3 & 0.5 \\ 0 & 0.1 & 0.5 \end{bmatrix}$$

and

$$P^2 = \begin{bmatrix} 0.5 & 0 & 0 \\ 0.4 & 0.5 & 0 \\ 0.1 & 0.5 & 1 \end{bmatrix}$$

The first model assumes that the infestation will worsen only gradually and may improve on its own. The second model assumes that the infestation will spread rapidly and will not improve on its own.

A script file `pests_adaptive1` is contained in the `mdpdemos` subdirectory. The first part of this file defines the problem parameters.

```
p=20;          % number of belief state values
delta=0.95;    % discount factor
D=[0;10;20];   % damage costs
Ct=10;         % treatment cost
T=inf;         % time horizon

% reward matrix (negative cost matrix)
R=-[D D+Ct];
% treatment transition matrix
Pt = [0.9 0.8 0.6
      0.1 0.2 0.4
      0.0 0.0 0.0];
% non-treatment transition matrix - model 1
P1 = [0.9 0.6 0.0
      0.1 0.3 0.5
      0.0 0.1 0.5];
% non-treatment transition matrix - model 2
P2 = [0.5 0.0 0.0
      0.4 0.5 0.0
      0.1 0.5 1.0];
```

The script file next calls `amdp` to create the transition and reward matrices defined for the expanded state space using $p = 20$ belief state intervals (21 values). Once those are obtained it sets up the model structure and calls `mdpsolve`.

```
% set up the belief state problem
[b,Pb,Rb,Svalsb]=amdp(p,{[Pn1 Pt],[Pn2 Pt]},R);
clear model
model.R=Rb;
model.P=Pb;
model.discount=delta;
model.Svals=Svalsb;
% call basic solver
results=mdpsolve(model);
```

Examination of the results reveals that it is never optimal to treat when there is no infestation, always optimal when the infestation level is high and optimal when the infestation level is medium if the degree of belief in model 1 is below 53%. The `mdpdemos` subdirectory also contains a demo file `pests_adaptive2` that adds a third model and solves the resulting adaptive management problem.

12.2 Partial Observability

Uncertainty may also arise because it is not possible to observe the states. The Partial Observability Markov Decision Problem (POMDP) is one way to address this issue. A POMDP redefines the states in terms of a set of beliefs about the value of the original states.

Each period new information arises in terms of some observable variable Y, which is correlated with one or more of the state variables. This necessitates defining a new set of probability matrices that relate Y to A and either to S or to S^+. Traditionally POMDPs define a probability matrix Q as

$$Q_{ijk} = Prob(Y = y_k | S^+ = s_j, A = a_i)$$

(note that the i subscript refers to the action and not to the current state or state/action combination). The new information, therefore, directly relates to the updated state. This would be the correct way to model a situation in which the new information arises during or after the transition to the new state.

An alternative is that the probability matrix Q is conditional on the current state

$$Q_{hik} = Prob(Y = k | S = s_h, A = a_i)$$

This would be the correct way to model a situation in which the new information arises before the transition to the new state. In both case, the actions must be independent of the current (unknown) state and hence there must be the same number of actions for every value of the state.

As with Adaptive Management, by defining the problem in terms of a set of belief states, it is converted to a standard MDP. An important difference between Adaptive Management and POMDPs, however, is that in AM the belief states augment the original states, whereas in POMDPs, the belief states replace the uncertain original states.

In a POMDP the future belief state is defined using Bayes Rule. If the information variable is defined conditionally on the future state then the updating rule for the belief state is

$$b_j^+ = Prob(S^+ = s_j | Y = y_k, A = a_i, b) = \frac{\sum_h Q_{ijk} P_{hij} b_h}{\sum_h \sum_j Q_{ijk} P_{hij} b_h}$$

and the probability that information state k will occur is

$$Prob(Y = y_k | A = a_i, b) = \sum_h \sum_j Q_{ijk} P_{hij} b_h$$

(in a change of notation from what has been used previously here $P_{hij} = Prob(S^+ = s_j | S = s_h, A = a_i)$)

On the other hand, if the information variable is defined conditionally on the current state then the updating rule for the belief state is

$$b_j^+ = Prob(S^+ = s_j | Y = y_k, S = s_h, A = a_i, b) = \frac{\sum_h Q_{hik} P_{hij} b_h}{\sum_h Q_{hik} b_h}$$

and the probability that information state k will occur is

$$Prob(Y = y_k | A = a_i, b) = \sum_h \sum_j Q_{hik} P_{hij} b_h$$

To make this operational, the belief space is discretized using p subintervals for each belief variable. The updating rule does not guarantee that the updated belief falls on one of these grid points and therefore an interpolation rule must be used to define the transition probabilities for the belief states. Here there are

$$\frac{(p + n^s - 1)!}{p\,!(n^s - 1)!}$$

feasible grid points and, as with Adaptive Management, triangularization is used for interpolation. Any point in the belief space can be interpolated using the values at the n^s nearest nodes. This results in a "probability" matrix with exactly n^s non-zero elements per node. This method of interpolation is implemented in the function `simplexbas`; the grid is initialized by calling `simplexgrid`.

Transition matrices as well as modified reward functions and state values can be obtained using the procedure `pomdp`. This procedure accepts as inputs p (the number of belief intervals for each state value), P, Q and R and returns the set of belief states b and the updated versions of P and R. If it is passed S, X and $\mathcal{I}^x$ these will be updated as well.

Example: Pest Infestation

A model developed in [1] for addressing pest infestation problems that involved partial observability illustrates the approach. The problem facing managers is twofold, whether to treat a potentially infested site and whether to monitor the site to determine the extent of the infestation.

Suppose that there a three levels of infestation: none, medium and high. Damages in the three states are calculated to be 0, 10 and 20, respectively. The site can be treated for the infestation, monitored, both or no action can be taken. There are, therefore, four possible actions: 1) do nothing, 2) monitor only, 3) treat only and 4) treat and monitor. Monitoring is assumed to cost 4 and treatment to cost 20. The complete reward matrix is

$$R = \begin{bmatrix} 0 & 0 & -20 & -20 \\ -10 & -10 & -30 & -30 \\ -20 & -20 & -40 & -40 \end{bmatrix}$$

The discount factor is assumed to be 0.95.

The monitoring does not effect the actual transition probabilities. If no treatment is undertaken the transition matrix is

$$P^n = \begin{bmatrix} 0.8 & 0.0 & 0.0 \\ 0.2 & 0.8 & 0.0 \\ 0.0 & 0.2 & 1.0 \end{bmatrix}$$

and if treatment is undertaken it is

$$P^t = \begin{bmatrix} 0.9 & 0.8 & 0.6 \\ 0.1 & 0.2 & 0.4 \\ 0.0 & 0.0 & 0.0 \end{bmatrix}$$

If monitoring is not undertaken the manager nonetheless receives some indication of which state exists. The probabilities associated with this variable are

$$Q^n = \begin{bmatrix} 0.5 & 0.5 & 0.0 \\ 0.3 & 0.4 & 0.3 \\ 0.1 & 0.4 & 0.5 \end{bmatrix}$$

Note that it is rows that sum to 1 because Q_{ij} expresses the probability that the signal is Y_j when the true state (next period) is S_i.

If monitoring is undertaken it is assumed that the manager knows precisely what the state is. This implies that Q^m is the identity matrix

$$Q^m = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

One should be clear about what this means. In this model a treatment is undertaken based on the current belief state. At the end of the period the signal is noted and the beliefs are updated. Therefore if monitoring is undertaken the belief state when the next action is taken will be perfect knowledge, i.e., the belief state will be one of $[1\ 0\ 0]$, $[0\ 1\ 0]$ or $[0\ 0\ 1]$.

The file `pests_pomdp.m` solves this model using 500 belief intervals for each state. Running the demo shows that the optimal control is to to do nothing when the probability that there is no infestation is high, to monitor when there is a medium probability of no infestation and to treat if the probability of infestation is high. It is never optimal to both monitor and treat.

12.3 Category Count Models

Category count models apply to situations in which there are a fixed collection of N separate items. Each item is classified into one of n^s categories and evolves as a controlled Markov chain with transition probability

$$Prob(s^+ = j | x = i) = P_{ij}(X)$$

The manager makes a decision concerning which of a set of treatments is applied to each item. Here s and x refer to the category and category/treatment combination for an individual item

and X is an vector indicating the number of items in each state/action combination (the sum of X over all its elements must equal N).

In such problems the state variable S is an n^s-vector of category counts and the state/action variable is an n^x-vector that gives the counts for each feasible state/action combination.

It is, of course, possible that P is constant, i.e., that each site behaves independently of all of the other sites. In this case P should be specified as a transition probability matrix. If P is dependent on X it should be coded as a function that accepts the n^x-vector X and returns an $n^s \times n^x$ transition matrix.

The function `catcountP` accepts a transition matrix or function for an individual item and returns the transition matrix for the n^s-vector of category counts. This problem can be grow very large very fast, even for relatively small N. In many situations not all state/action combinations are feasible and these should be eliminated. For example, it may only be possible to apply action 2 to, at most, 5 out of the N sites. There are (at least) three possible strategies to address the size issue. One strategy replaces small probability values with 0 so sparse matrix methods can be used. The second is to compute the probability values as needed using the `EV` feature (see Section 8.2). This could be time consuming but would address the storage problems. If the actions operate by changing the number of items in each category, the `Iexpand` feature (see Section 8.1) can be used to avoid computation and storage of redundant probabilities.

The script file `pests_catcount` implements a category count model. Consider once again a pest infestation problem but one with $N = 25$ sites and a limited treatment budget of $B = 100$. The per site treatment costs depend on the severity of the infestation: $C = [6\ \ 20\ \ 30]$ and the per site damage costs are $[0\ \ \ 20\ \ \ 50]$. Two cases are considered. The first has each site evolving independently of the others with fixed per site transition probabilities P_1 if no treatment is applied and P_2 if a treatment is applied.

The second case has colonization probabilities (i.e., a site that changes from uninfested to moderately infested) that depend on the number of infested sites according to

$$c = \frac{1}{1 + e^{\beta_0 + \beta(X/N)}}$$

The values of the β_0 is a scalar and β is a vector. The values of $\beta 0$ and β depend on whether the site is treated or not. Note this is change only affects the first 2 elements in the first row of the transition matrices. The function that generates the probabilities is coded as the function file `pests_catcountfunc`. A function to pass to `catcountP` is created using

```
Pfunc=@(Xj) pests_catcountfunc(Xj,b0u,b0t,b1u,b1t,P1,P2,N);
```

The complete set of feasible state/action combinations can be obtained using

```
X=double(simplexgrid(6,N,N,1));
ind=X*[0;0;0;C']<=B;
X=X(ind,:);
```

Here X has six variables (columns). The first line gets all possible state/action combinations with the first three indicating the number of sites for each category not receiving treatment and columns 4-5 the number in each category receiving treatment. The second line identifies the state/action combinations that do not exceed the budget and the third line extracts those

combinations. The values of the state and the state index can then be obtained using

```
[S,temp,Ix]=unique(X*[eye(3);eye(3)],'rows');
```

The `unique` function puts the values of S into lexicographic ordering; its third output is an index that associates the rows of the input matrix with rows of S.

The transition matrix can now be obtained. For the first case with P_1 and P_2 constant one can use

```
P=catcountP(N,3,6,P',X');
```

(`catcountP` currently requires that the transition matrices and the matrix of state/action combinations are passed in column stochastic form). For the second case with non-constant transition matrices one can use

```
P=catcountP(N,3,6,Pfunc,X');
```

The script file uses `mdpplot` to display the optimal decision rule. It also uses `longrunP` to compute the long-run joint distribution. These tools are described in the next section.

13 Tools for Analysis of Solution Results

MDPSOLVE contains a number of features to facilitate analysis of solution results.

13.1 Expected Time Paths

To understand how a model behaves over time a useful tool is to plot the paths of the expected evolution of a variable over time. Suppose that one is interested in the behavior of $f(S)$. The desired time paths can be computed using

$$Z_{jh} = E[f(S_{t+h})|S_t = s_j] = [P^h f(S)]_j$$

Thus Z_{jh} is the expectation of $f(S_{t+h})$ conditional on the current S being equal to s_j.

The procedure `epath` implements this. It's calling syntax is

```
Ef=epath(fs,P,Ix,T,colstoch)
```

where `fs` is an n^s-vector of values of the function f evaluated at each of the n^s values of S, P is an $n^s \times n^x$ transition probability matrix, $\mathcal{I}^x$ is an n^s-vector of indices that define a decision strategy (only non-random strategies are supported) and T is the number of time periods (the maximum value of h desired). The last input, `colstoch` is optional; if omitted it is assumed to be 1. If P is passed in row stochastic form set `colstoch` to 0.

This procedure also works when P is passed as an $n^s \times n^s$ matrix with `Ix` set to empty. For example, using the `pstar` matrix returned by `mdpsolve` is the same as using `model.P` and the value of `Ixopt` returned by `mdpsolve`. Thus `epath` can be used to generate both optimal and suboptimal paths.

This procedure is demonstrated in the script file `epathdemo`. The demo defines a category count model with 2 categories and $N = 10$ elements. Each element evolves according to the transition matrix

$$p = \begin{bmatrix} 0.8 & 0.1 \\ 0.2 & 0.9 \end{bmatrix}$$

The state variable is the number of successes (the number of elements in category 2). It plots the evolution of the number of successes over time, demonstrating that, regardless the initial state, the expected number of successes converges to a common value of 6.66.

Note that the n^s time paths computed by `epath` each begin at one of the state values. If the state is, in fact, continuous, the time path for a non-grid value starting point can be found by interpolation. If the state variables are defined on a rectangular grid use `rectbas`. If they are defined on a simplex use `simplexbas`.

13.2 Simulated Time Paths

It is often useful to simulate time paths of the state and action variables to obtain a sense of how the model behaves over time. The utility `mdpsim` provides simulated time paths. It's syntax is

```
[S,A] = mdpsim(s0,P,Ix,T,colstoch)
```

The first input is a k-vector of initial state indices. `mdpsim` will create k separate (and independent) time paths with the ith path beginning at the state indexed by the ith element of `s0`.

The second input is an $n^s \times n^x$ transition probability matrix (or $n^m \times n^s$ if `colstoch` is set to 0). The third input is an index vector that determines which rows of P are used (for the optimal decision rule use the $\mathcal{I}^{x*}$ (`Ixopt`) output from `mdpsolve`). The fourth input is the number of time steps to be simulated. The fifth input is optional if P is in the $n^s \times n^x$ form and should be set to 0 if P is in the $n^x \times n^s$ (row stochastic) form.

As with `epath`, this procedure also works when P is passed as an $n^s \times n^s$ matrix with `Ix` set to empty. For example, using the `pstar` matrix returned by `mdpsolve` is the same as using `model.P` and the value of `Ixopt` returned by `mdpsolve`. It is, however, possible to evaluate other decision rules than the optimal one determined by `mdpsolve`.

The procedure is demonstrated with the script file `pests_sim`. It simulates and plots 3 paths for the state starting at each value of the state over 25 time periods. It also prints out the long-run probabilities of being in any one of the states and the estimated probabilities based on the simulated paths.

It is important to note that `Ix`, `s0`, `S` and `A` all are composed of index numbers for the states or state/action combinations and not the values of those variables. See Section 6.1 for a discussion of this distinction.

The simulation procedure `mdpsim` should not be used with models involving beliefs states (i.e., if P is created using `amdp` or `pomdp`). Simulation of such models is more involved and requires special handling for which no general procedures have been designed yet.

13.3 Long Run Analysis

For infinite horizon time-autonomous problems it is possible to compute the longrun probability distribution for the state variables, from which a number of interesting summary measures can be obtained. When it exists, the long-run distribution π, a $n^s \times 1$ vector, satisfies $\pi = {P_A}^\top \pi$, where P_A is an $n^s \times n^s$ probability matrix corresponding to a specific strategy (A).

One way to interpret the longrun distribution is as the limit of the transition matrix for time periods in the distant future. It is is easy to see that

$$Prob(S_{t+h}|S_t) = P^h$$

If we let $h \to \infty$ and P has a unique longrun distribution π, then P^∞ will be composed of n^s rows each equal to $\pi^\top$:

$$\lim_{h\to\infty} P^h = \begin{bmatrix} \pi^\top \\ \pi^\top \\ \cdots \\ \pi^\top \end{bmatrix}$$

To determine the longrun expectations of some function f of the state variables, simply compute $\pi^\top f(S)$. When there is more than one state variable, the marginal distribution of state variable i is given by

$$\pi^i_j = Prob(S_i = s_{ij}) = \sum_{k:S_{ik}=s_{ij}} \pi_k$$

The functions `longrunP` and `marginals` are available to compute long run probability information for infinite horizon, stationary ergodic models.

In analyzing the solution of a MDP, the long run distribution of the transition probability matrix associated with the optimal actions should be used; this is returned as `results.pstar`, . Note that `pstar` is always returned in column stochastic form (rows are future, columns are current states) even if the original data was in row stochastic form. This can be passed to `longrunP`, which, in turn, can be passed to `marginals`:

```
pi=longrunP(pstar);
M=marginals(pi);
```

Using the longrun distribution it is very easy to compute longrun expectations of functions of the state variables. Suppose one has specified an n^s-row matrix `S`, where the columns represent the various state variables. The expectation of the state variable is computed using `pi'*S` and the expectation of $f(S)$ is `pi'*f(S)`.

13.4 Graphics

Often figures are useful ways to present and interpret results. A plotting routine designed for MDPs is available with MDPSOLVE. This procedure, `mdpplot`, will create a plot of the value function or optimal decision rule for up to four state variables. It's basic calling syntax is

```
mdpplot(S,v,index,labels,options)
```

Here S is the n^s-row matrix of state variable values, v is an n^s-vector of values to be plotted (it could be the value function or the action vectors returned by `mdpsolve`).

The simplest way to use this procedure is to create a single plot in a problem with two state variables. In this case the first element of `index` gives the number of the state variable to be placed on the x-axis and the second element gives the number of the state variable that goes on the y-axis.

In more complicated problems with more state variables `mdpplot` can be used to create a set of subplots. This is especially useful if some of the state variables take on only a few discrete values. If `index` is a 3 vector a row of subplots will be created, one for each unique value of the variable indexed by the third element in the index vector. If the index vector is of length 4, a matrix of plots will be created with rows associated with the fourth variable listed and columns with the third variable listed. If the third element equals 0 a single column of subplots is created.

The input variable `labels` should be a cell array of the same length as `index` containing a set of strings for the variable names used for labelling the graph.

The `options` input is a structure variable that is used to control certain aspects of the plot's appearance. Possible fields are:

- **figuretitle** : a title placed in the border of the figure window
- **edges** : 1 to plot edge borders around each cell
- **squareplot** : 1 to make the x and y axes of equal size

- **addlegend** : 1 to add a legend
- **vertical** : 1 to make the legend have vertical orientation (placed outside to the east - if 0 placed to south)
- **colorbartype** : 1 to make the legend be a colorbar type rather than discrete
- **legendlabels** : a cell array of labels for a legend
  there must be at least as many as there are unique values of V
- **legendtitle** : a string placed above the legend to indicate the name of the plotted variable
- **grayscale** : 1 to use gray color scheme
- **noticklabels** : 1 to suppress tick labels on x and y axes

The placement and size of the various components of the figure generated by `mdpplot` may be altered if desired. First, one can change the relative size of some components by altering the size of the figure window. Second, one can click on the white arrow (Edit Plot) button and use the mouse to select, move and size the components.

14 Troubleshooting

As with any computer software, stuff happens. This section is designed to help you through some common problems.

14.1 Memory Issues

MATLAB can be a memory hog. To understand why, consider a simple statement like `E=A*B+C*D;`. First, this require that all of the input matrices A, B C and D have to be in memory. Then memory is set aside for AB and more memory for $C * D$. Then memory needs to be obtained to hold the result. The reason for all of this memory is that MATLAB processes arrays. This means that three arrays must be created to obtain one result.

If one want to be sure that all of this memory is not created one could use the following: `E=A*B; E=E+C*D;` The reason this might work is that recent versions of MATLAB will overwrite the matrix E in the second statement (this will not happen, however, if E is sparse).

Consider another common situation arising using a statement like `P=[P1 P2];` Here P_1 and P_2 are already in memory and concatenating them requires space for the combined matrix, essentially requiring twice as much memory. Even if the command works you are storing the same data in two different places.

The easiest "fix" is to be ruthless in clearing from memory any arrays that are no longer needed. For example using `P=[P1 P2]; clear P1 P2;` leaves only the combined matrix in memory. In our first example one could use

```
E=A*B; clear A B; E=E+C*D; clear C D;
```

This can be done with large arrays that are subsequently redefined. For example `C=A+B; A=C+B;` might benefit from using `C=A+B; clear A; A=C+B;` (might because MATLAB is gradually improving its memory management and it is not always clear what is going on beneath the surface without extensive sleuthing).

Working with large sparse matrices can prove especially challenging because sparse matrices often need to be copied from one location to another in order to accommodate increased growth. For example, suppose one computes a matrix a column at a time in a loop:

```
P=zeros(m,n);
for j=1:n
  P(:,j)=getPj(j);
end
```

Here P is a full (non-sparse) matrix and memory to hold it is preallocated before the beginning of the loop. In this case no new memory is needed to hold P. Consider on the other hand

```
P=sparse(m,n);
for j=1:n
  P(:,j)=getPj(j);
end
```

Here P is initialized as a sparse matrix but the `sparse` command as used here does not allocate any memory to hold the matrix. Instead the memory is reallocated every time through the loop and the matrix thus far created is moved.

If you knew how much memory would be needed one could use instead

```
P=sparse([],[],[],m,n,memneeded);
for j=1:n
  P(:,j)=getPj(j);
end
```

Often the amount of memory need is not known ahead of time but a reasonable guess might be made. One could do the following, for example

```
P=Pj(1);
P=sparse(P,1:m,1,m,n,nnz(P)*n);
for j=2:n
  P(:,j)=getPj(j);
end
```

This gets the first column of P and estimates the memory requirements by multiplying the number of non-zeros in that column by the number of columns. This allocates exactly the right amount of memory if all of the columns have the same number of non-zeros.

The thing to avoid (generally, because there are exceptions) is to build matrices through concatenation. Consider the following code

```
P=Pj(1);
for j=2:n
  P=[P getPj(j)];
end
```

This produces an identical matrix to the one created above. In this case, however, the matrix is resized at every step of the loop. This means that new memory is allocated at every step and the old matrix copied into the new one before the new column is added. In general this will use much more memory and also be really slow.

15 Technical Appendix

15.1 Interpolation on a Simplex

Any point in a simplex is a set of q non-negative numbers that sum to a constant C. We can define a grid of points by subdividing the interval from 0 to C into p evenly spaced intervals. Any point can then be normalized by multiplying it by C, giving us a q-vector of non-negative integers that sum to p.

In order to interpolate we must have a way of mapping the grid points into their place in the ordering of points. Here we use a lexicographic ordering. To determine the index of a grid point v we can use the following approach. First define the multiset coefficient

$$T(i,j) = \frac{(i+j-1)!}{i!(j-1)!} = \binom{i+j-1}{i} = \left(\!\!\binom{i}{j}\!\!\right)$$

These value satisfy the recursive relationship $T(i,j) = T(i-1,j) + T(j-1,i)$ with $T(i,1) = T(1,j) = 1$. The index value associated with a grid point v is

$$I(v) = 1 + \sum_{i=1}^{q-1} T(\eta_{i-1}, q-i) - T(\eta_i, q-i)$$

where $\eta_0 = p$ and $\eta_i = \eta_{i-1} - v_i$ for $i > 0$ (if 0-based indexing is used the addition by 1 is not needed).

This can be simplified using the recursive definition of T:

$$I(v) = T(p,q-1) - \sum_{i=1}^{q-1} \Big(T(\eta_i, q-i) - T(\eta_i, q-i-1) \Big) = T(p,q-1) - \sum_{i=1}^{q-1} T(\eta_i - 1, q-i)$$

(this implicitly uses the fact that $T(i,0) = 1$). One issue with this method arises when $\eta_i = 0$ because $T(-1,j)$ is not defined. This situation arises, however, when all of the remaining values of v_i are 0; at this point the summation can be stopped.

An example will illustrate the method. Suppose that $q = 4$ and $p = 3$. The index values and the vertex values are given by:

index	v_1	v_2	v_3	v_4
1	0	0	0	3
2	0	0	1	2
3	0	0	2	1
4	0	0	3	0
5	0	1	0	2
6	0	1	1	1
7	0	1	2	0
8	0	2	0	1
9	0	2	1	0
10	0	3	0	0
11	1	0	0	2
12	1	0	1	1
13	1	0	2	0
14	1	1	0	1
15	1	1	1	0
16	1	2	0	0
17	2	0	0	1
18	2	0	1	0
19	2	1	0	0
20	3	0	0	0

The multiset coefficients needed are given by

	q			
p	0	1	2	3
0	1	1	1	1
1	1	2	3	4
2	1	3	6	10
3	1	4	10	20

Suppose that we are interested in finding the index number of the vertex $[1\ \ 0\ \ 1\ \ 1]$. The values of the η_i are 2, 2 and 1 leading to an index of

$$I([1\ \ 0\ \ 1\ \ 1]) = T(3,3) - \Big(T(1,3) + T(1,2) + T(0,1)\Big) = 20 - (4+3+1) = 12$$

It is easily verified that vertex 12 in the above table is indeed $[1\ \ 0\ \ 1\ \ 1]$.

To illustrate the situation when η_i becomes 0, consider the vertex $[1\ \ 2\ \ 0\ \ 0]$. Here the η_i are 2, 0 and 0. If we halt the summation when η_i becomes 0 the index is simply

$$I([1\ \ 1\ \ 0\ \ 0]) = T(3,3) - T(1,3) = 20 - 4 = 16$$

In MATLAB this can be written

```
ind=tab(p+1,q-1);
eta=p;
```

```
for i=1:q-1
  if v(i)>eta, break; end
  eta=eta-v(i);
  ind=ind+tab(eta,q-i);
end
```

Thus far we've just discussed how to determine where in a lexicographic ordering a given vertex is located. The interpolation method used must determine which vertices receive non-zero weights and the values of the weights.

MDPSOLVE implements the modification of the Freudenthal triangulation method discussed by [3] and [6] in the context of solving POMDP problems. As these articles are readily available the method is not repeated here. To increase speed and reduce memory usage for these low level functions, MDPSOLVE has a set procedures coded in C to handle grid functions.

16 List of MDPSOLVE Procedures

This section serves as a reference for the various MATLAB tools used in MDPSOLVE.

Tools Used for Working with Matrices of Variable Values

- index - converts between index numbers and subscript vectors
- rectbas - forms an interpolation matrix for regular (rectangular) grids
- rectgrid - forms a regular (rectangular) grid
- rectindex - finds the index number for values on a regular (rectangular) grid
- simplexbas - form interpolation matrix for a grid on a simplex (used to interpolate belief states and count variables)
- simplexgrid - forms a grid for a simplex (used to define grids over beliefs and count variables)
- simplexindex - finds the index number for values on a simplex grid

Basic Modeling and Optimization Procedures

- amdp - converts an adaptive MDP to standard form
- catcountP - converts a transition matrix for an individual item into a category count transition matrix
- g2P - computes a transition probability matrix from a transition function
- mdpsolve - the basic solver for Markov Decision Problems (MDPs)
- pomdp - converts a Partial Observable Markov Decision Problem (POMDP) to standard form Model

Tools for Solution Analysis and Display

- epath - computes time paths of functions of the state variable
- longrunP - computes long-run probabilities from a transition probability matrix
- marginals - computes the marginal distributions of a set of variables
- mdpplot - plotting function for value functions and optimal actions
- mdpsim - simulates time paths of a controlled Markov process

17 A Template for Coding in Matlab

Solving problems using MDPSOLVE is best and most easily accomplished by using a template that follows a well defined set of steps. These steps are:

- name and assign values to problem parameters
- specify the state values and state/action combinations
- specify the reward vector
- specify the transition probability matrix
- define the model variable
- pass the model variable to `mdpsolve`
- interpret the output from `mdpsolve`

Problem Parameters

Generally this consists of a set of lines such as

```
c=0.5 % cost parameter
```

This line creates a variable called `c` and assigns it the value 0.5. Parameters may take the form of vectors or matrices, which can be defined by listing their values (if they are small enough). For example

```
D=[0; 5; 20];  % damage costs
```

creates a 3×1 vector of values. A good introduction to creating variables, including vectors and matrices can be obtained by opening the MATLAB documentation (by typing `doc` at the MATLAB command line) and reading the pages starting in the section

```
Mathematics/Matrices and Arrays
```

Another way to assign values to a vector that is often useful is to create a vector that contains a set of equal spaced values between a lower and an upper bound. There are two ways to do this. This first uses the ":" (colon) operator:

```
w=0:4
```

creates the 1×5 vector $[0\ \ 1\ \ 2\ \ 3\ \ 4]$.

```
w=0:2:4
```

creates the 1×3 vector $[0\ \ 2\ \ 4]$. the general form is:

beginning value:step size:ending value

If the step size is omitting it is assumed to be 1, as in the first example.

This format can be a bit ambiguous as can be seen in the following example:

```
w=0:2:5
```

Does this result in $[0\ \ 2\ \ 4]$ or in $[0\ \ 2\ \ 4\ \ 6]$. The problem here of course is that the "ending value" is not equal to the starting value plus an integer multiple of the step size. A less ambiguous command is

```
w=linspace(0,4,3);
```
which creates $[0\ \ 2\ \ 4]$. Here is syntax is
```
w=linspace(starting value, ending value, number of values)
```

Both the colon operator and `linspace` create row vectors. If a column vector is desired use the transpose operator, e.g.,
```
w=linspace(0,4,3)';
```
You may also want to define simple functions at this stage, which are much like parameters. For example we can use what is known in MATLAB as an anonymous function:
`logit = @(x,beta) 1./(1+exp(-(x*beta)));` This defines the function

$$f(x) = \frac{1}{1 + \exp(\sum_i x_i \beta_i)}$$

It requires that x and β be compatible arrays; in this case vectors of the same size with x a row vector and β a column vector. One of the most frequent run-time errors in a language like MATLAB is that matrix arithmetic operations are attempts on matrices of the wrong size.

Notice that each line ends with ";" (semi-colon). This is not necessary but it prevents the value of the variable from being written to the screen, which can be downright annoying. Also notice that I have put a comment after the assignment. In MATLAB any text following the "%" sign in a given line is interpreted as a comment. It is very useful to put comments in your code so you remember how to interpret it and so others can read it.

States and State/Action Combinations

It is generally a good idea to create a list of all of the values of the state and to put in a comment describing how the values are interpreted. In many cases the values assigned are arbitrary; for example 1 might be active and 2 might be inactive (you might, however, want 0 to represent inactive and 1 to represent active).

Anything with a nominal "scale" can be assigned values arbitrarily. In general it will make life easier if you use the values 1 through n that have a ordinal scale with n possible values. For example low, medium and high would typically be assigned the values 1, 2 and 3 or, possibly, 0, 1 and 2. For variables that are measured with a cardinal scale one should use the actual values. The scale of the values can be important as well. It is generally best to use numbers that are near 1 in value so it is better, for example, to express a population size in millions of animals rather than in individual units. This reduces the effect of roundoff error that arises in doing numerical woek.

In many cases the values of the state can be assigned in the same way as a vector of parameter values (for instance by using the colon operator or the `linspace` procedure). The same goes for action variables.

It is also a good idea to list all of the state/action combinations. For example if there are three states (1, 2 and 3) and two actions for each state (0 and 1) the complete set of state/action combinations is

1 0
1 1
2 0
2 1
3 0
3 1

The state/action combinations could be listed in any order but it is highly desirable to keep things orderly so it is easy to find and interpret the results. The ordering shown here and the one that is used generally by MDPSOLVE is the so-called lexicographic ordering, which sorts first on the first column and then on the second column, etc.

The easiest way to obtain this table (matrix) of values is to use the `MDPSOLVE` procedure `rectgrid`. For example, the states actions and state/action combinations just described can be obtained using

```
S=[1;2;3];
A=[0;1];
X=rectgrid(S,A);
```

Notice that S and A are both defined to be column vectors; this is necessary because `rectgrid` handles row vectors differently from column vectors.

It may also be useful to create the variables n^s and n^x to store the number of states and the number of state/action combinations. This can be accomplished in a number of ways but perhaps the easiest and most general is to use `ns=size(S,1)` and `nx=size(X,1)`, which assigns the number of rows of `S` to `ns` and the number of rows of `X` to `nx` (the 1 refers to the first dimension of the array, which is the rows; a 2 would refer to the columns).

It is also necessary to inform the solver as to which states are associated with each state/action combination. This is done by defining an n^x-vector `Ix` composed of integers between 1 and n^s.The easiest way to do this is to use the utility `getI` which uses the syntax

```
[Ix,S]=getsaind(X,statevars);
```

The second input to this utility is a vector listing which columns of X contain the state variables. $\mathcal{I}^x$ can then included as a model field to be passed to `mdpsolve`.

In the example given above one would use

```
Ix=getI(X,1);
```

This case is trivial as `Ix` and the first column of X are identical. The second output may be used to define S or as a check to ensure that S is defined compatibly with `Ix` (i.e., that it is arranged lexicographically).

Specify the Reward Vector

Rewards are specified as vectors of length n^s and are functions of the state/action combination X. Typically the easiest way to assign values to rewards is either to enumerate the values, e.g.,

```
R=[1 2 4 2 3 6]';
```

or to compute them as a function of X, e.g.,

```
R=p*X(:,2)-c*X(:,1);
```

(this is the function $pA - cS$ since the second column of X is the value of the action and the first column is the value of the state).

Sometimes it we want to use different functions for different actions. For example, suppose there are two actions with action 1 having a reward of 0 and action 2 having a reward of S. This could be accomplished in a variety of ways but perhaps the simplest is to use

```
R=X(:,1).*(X(:,2)==2);
```

The term in `X(:,2)==2` defines an n^s vector of 0s and 1s, with the 1s associated with state/action combinations in which the action equals 2. This is then multiplied by the value of the state, which is in the first column of X. Note the use of the ".*" operator, which performs element-by-element multiplication.

Specify the transition probability matrix

This is often the most challenging part of coding a problem mainly because there are so many ways that P can be defined. For small problems one can simply list all of the values, as was done in the simple pest management problem (in Section **??**). The most general way to specify P is to use for loops, with loops over the current state/action combinations (1 through n^s) or over the future states (1 through n^s) or both. Loops tend to execute more slowly in MATLAB so specifying a whole row or column of P with a single command is usually preferable to using a double loop.

Anytime loops are used it is a good idea to pre-allocate P so that enough memory is set aside to hold the whole matrix. The easiest way to do this uses the command

```
P=zeros(ns,nx);
```

This creates an $n^s \times n^x$ matrix that has all 0 elements. For very large problems in which many of the elements are actually 0, it may be preferable to use a sparse matrix storage format. A sparse data format is one in which only the non-zero elements, along with row and column indices, are kept. If the fraction of non-zero elements is relatively small, the sparse data format requires less memory and can be processed faster than a full (or dense) matrix format. To create an empty sparse matrix one can use the syntax

```
P=sparse(ns,nx);
```

This also creates an $n^s \times n^x$ matrix. It does not, however, allocate memory to hold the non-zero elements. If you know that P contains q non-zero elements (even if this is only an approximation) it is better to use the syntax

```
P=sparse([],[],[],ns,nx,p);
```

Once P is pre-allocated one implements the loop. A loop over the rows (the current state/action combinations) looks like

```
for j=1:nx
  P(:,j) = ...;
end
```

where the `...` must be filled in with the specific probabilities relevant for the problem. Loops over columns (futures states) look like

```
for i=1:n
```

```
  P(i,:) = ...;
end
```

Loops over both rows and columns look like

```
for i=1:ns
  for j=1:nx
    P(i,j) = ...;
  end
end
```

It is good programming practice to use indentations to help make clear the loop structure. The MATLAB editor will automatically do the indentation for you (use File/Preferences/Editor/Tab to set the defaults).

The `sparse` procedure can also be used with a set of row indices, columns indices and values. For example, suppose that the state takes on n^s evenly spaced values between 0 and $\bar{S}$:

```
S=linspace(0,Sbar,ns)';
```

and, for a given action, the probability is 0.5 that the state will move up one value and 0.5 that it will move down one value (except at the end-points of course). This can be obtained in the following way:

```
rowind=[1:ns 1:ns];
colind=[1 1:ns-1 2:ns ns];
P=sparse(rowind,colind,0.5,ns,ns);
```

This creates and $n \times n$ matrix that looks like

$$\begin{bmatrix} 0.5 & 0.5 & 0 & \dots & 0 & 0 & 0 \\ 0.5 & 0 & 0.5 & \dots & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 0.5 & 0 & 0.5 \\ 0 & 0 & 0 & \dots & 0 & 0.5 & 0.5 \end{bmatrix}$$

The rest of the stuff

The procedure `mdpsolve` requires that the problem specification be bundled in a structure variable that has a set of fields for the different elements of the problem specification. The model structure can be created in one of two ways. The first ways uses the syntax

```
clear model
model.P=P;
model.R=R
model.d=delta;
```

Here we first clear the model variable to ensure that if there is any previously defined variable called `model` it will not interfere with the new one being defined. Then values are assigned to the various fields. The alternative uses the `struct` procedure:

```
model=struct('P',P,'R',R,'d',delta);
```

This syntax works well unless one of the variables assigned to the structure is a cell array, in which case unexpected results will occur (we will assign cell arrays to some fields when we define models with stages; for more information see Chapter 11).

Once the fields are assigned, the `mdpsolve` procedure is called:

```
results=mdpsolve(model);
```

This returns a structure variable with various fields that describe the results from the solver. The first, `v`, is the value function and the second, `Ixopt`, is an index vector specifying the optimal strategy (`Ixopt` contains integer values between 1 and n^x). When the action values are listed in the second column of X, the action values associated with the optimal strategy can be obtained using `A=X(results.Ixopt,2)`.

18 MDPSOLVE Reference

The `mdpsolve` procedure is the basic solver used to find optimal actions for a given problem. It is used by setting up two structure variables, called `model` and `options`. Structure variables in MATLAB are essentially containers in which information can be defined and accessed by field names.

The model structure describes the problem by defining (at a minimum) the rewards, the transition probabilities and the discount rate. Other fields allow more complicated models to be defined.

The options structure defines a number of fields that control the operation of the procedure. These include a number of fields controlling problems with infinite horizons as well whether information is printed to the screen as the procedure executes and precisely what information the procedure returns to the caller.

18.1 Fields for the model structure

- R or reward : information about the current rewards for each state and action. It can be an n^x-vector or an $n^s \times n^a$ matrix (where $n^a = n^x/n^s$). May also be a cell array of reward vectors or matrices (for models with stages) or a function to calculate them.

- P or transprob : information about the transition probability matrices. There are many ways to define this. It may be a matrix, a cell array of matrices or a function to calculate them.

- d or discount: discount factor or n^s-vector of discount factors. May also be a cell array of discount factors or a function to calculate them.

- T or horizon : the time horizon for the problem (set to ∞ for infinite horizon problems) [default=∞].

- ns : number of state variables - it may be possible to determine this from the other input data but sometimes it is not possible and it never hurts to include it to ensure model validity. May be an n^{stage} vector or cell array

- Ix : an n^x-vector associating state/action combinations with specific states. Each element of $\mathcal{I}^x$ should be an integer between 1 and n^s (the number of state). Can also be an n^{stage} element cell array of $\mathcal{I}^x$ vectors.

- Iexpand: an n^x-vector associating state/action combinations with columns (or rows if colstoch=0) of P. If P has columns that are identical for alternative state/action combinations, one can reduce processing time and memory usage by defining a P that contains only unique columns and using `Iexpand` to associate these columns with various state/action combinations (see discussion in Section 8.1). Can also be an n^{stage} element cell array of $\mathcal{I}^{expand}$ vectors.

- colstoch: set to 0 if rows of transition matrices represent current state/action combinations and columns represent future states. Set to 1 if the converse is true. Not needed if it can be determined by other input data.
- EX: set to 1 if P is a function that accepts an n^s vector and returns an n^x vector.
- nstage: number of stages (a positive integer). The default is 1.
- nrep: the numbers of repetitions per stage - an nstage-vector of positive integers. The default is a vector of 1s.
- vterm : terminal value. An n^s-vector; only used if $T < \infty$ [default: n^s-vector of 0s]
- X: n^x-row matrix of the values of the state/action combinations [default: none]
- svars: a vector containing the column numbers of X associated with state variables
- name: a string variable that will be printed on the summary report (especially useful for identifying results when multiple models are used)

Of these fields `R`, `P`, and `discount` are required. If the `T` field is omitted it is assumed that $T = \infty$. If the `vterm` field is included then the `T` field must also be specified with $T < \infty$ and integer. If the `Ix` field is omitted then either X and svars must be included (these will be used to obtain $\mathcal{I}^x$ or there must be the same number of actions for each state and R must be defined as an $n^s \times n^a$ matrix (see the discussion in Section 5).

The fields nstage and nrep specify the stage structure of a problem (see Section 11 for a discussion). Both fields are optional if their values can be deduced from other input values. For models with stages the fields `R`, `P`, `discount`, `Ix` and `Iexpand` can all be defined as cell arrays with `nstage` elements.

The `X` field is used only when the `Ix` is omitted and when `R` is a vector. If these two situations exist `X` and `svars` are used to determine $\mathcal{I}^x$. The function `mdpsolve` itself does not make any use of the values of the state or action variables, using only the index values. The main use of the variable values is to convert problems with continuous variables into discrete problems and to interpolate state transitions defined be functions rather than by probability matrices (for more on the distinction between indices and values see the discussion in Section 6.1).

As a general rule MDPSOLVE attempts to determine the values of fields that are not included in the model structure. If it is unable to do this it will result in an error message. If this happens you should add the relevant information to the model structure and re-run `mdpsolve`.

The following table lists the possible values that the various fields can legitimately have. Other specifications will result in error messages and failure to solve the problem. The `results` variable returned by `mdpsolve` will contain a record of the errors that will be displayed if the `options.print` field is greater than 0 or if an `mdpreport(results)` command is issued.

field	numerical array			function handle	1 or n^{stage} element cell array	range
	scalar	vector	other			
T	Y					$\{1, ...\}$
nstage	Y					$\{1, ...\}$
colstoch	Y	n^{stage}			Y	logical
EV	Y	n^{stage}			Y	logical
ns	Y	n^{stage}			Y	$\{1, ...\}$
nx	Y	n^{stage}			Y	$\{1, ...\}$
nrep	Y	n^{stage}			Y	$\{1, ...\}$
vterm		n^s				$(-\infty, \infty)$
Iexpand		n^x			Y	$\{1, ..., k\}$
Ix		n^x			Y	$\{1, ..., n^s\}$
d	Y	n^s		Y	Y	(0,1]
R		n^x	$n^s \times n^a$	Y	Y	$(-\infty, \infty)$
P			*	Y	Y	$[0, 1]$

* P can be $n^s \times n^x$ (if `colstoch==1`) or $n^x \times n^s$ (if `colstoch==0`). Also allowed, if `Iexpand` is defined, are $n^s \times k$ (if `colstoch==1`) or $k \times n^s$ (if `colstoch==0`) where k is the maximal value in `Iexpand`.

Table 1: Valid Input Data for `model` Fields

18.2 Fields for the options structure

- print: print level (0) no display (1) display summary information (2) display summary and iteration information (for $T = \infty$) [default: 1]
- checks: `mdpsolve` performs a variety of consistency checks to ensure that a problem is well posed. If you are sure that these checks are unnecessary, set the `checks` field to 0. [default: 1]
- keepall: keep values and actions for every iteration (for $T < \infty$ only) [default: 0]
- algorithm: 'p' for policy iteration, 'f' for function iteration [default: 'p']
- v: starting value vector [default: n^s-vector of 0s]
- maxit: maximum number of iterations [default: $20 \ln(n^s)$ for policy iteration, $250 \ln(n^s)$ for function iteration]
- modpol: maximum number of sub-iterations that update the value function using the same policy (modified policy iteration)
- tol: convergence tolerance [default: 10^{-8}]
- nochangelim: stop if action doesn't change in nochangelim iterations [default: ∞]

- relval: determines the method used for non-discounted problems. relval=0 does nothing special (this does not work with policy iteration). relval$\in \{1, 2, \ldots, n^s\}$ uses the value of relval to determine which element of the value function is set to 0 using the relative value method.

- vanish: vanish$\in (0, 1)$ uses a "discount factor" equal to vanish but adjusts the value function to approximate the average expected reward. Set this close to but not equal to 1 to implement the vanishing discount method. Otherwise set to 0 or don't se this field.

The first two options apply to all problems. The `print` option is normally set to 1. This generates a summary report and will display any error or warning messages on the screen. Setting `print` to 2 causes `mdpsolve` to print progress information for each iteration to the screen for infinite horizon problems. This can be useful for monitoring the progress of the program and for troubleshooting if convergence issues arise. Setting it to 0 eliminates any writing to the screen. Solution details can be obtained from the `results` variable returned by `mdpsolve`. Using `mdpreport(results)` produces the summary report.

For finite horizon problems ($T < \infty$) the `keepall` option can be used to save and return the value function vector and optimal action vector for each time period. This can result in these being large arrays and the default is to only return the first period values. If the `keepall==1` option is used, the results are returned with the first column representing the current period and the last representing the time horizon date (T). Note that the terminal value defined by `model.vterm` is NOT included in the results.

The remaining options apply only to infinite horizon problems. These problems attempt to find a fixed point that solves the Bellman equation. The iterations require a starting vector (`v`) and then iterate until a convergence criteria is met or a maximum number of iterations (`maxit`) is exceeded. Two kinds of iteration can be used, function iteration and policy iteration. The default is to use policy iteration, which tends to be faster for moderately sized problems. Use the `algorithm` field to change this option. `modpol`, `tol` and `nochangelim` apply only to function iteration. `modpol` specifies the maximum number of sub-iterations that use the same policy (skipping the maximization step). There are two convergence criteria that are used by `mdpsolve` for function iteration (`algorithm='f'`). First, if the value function times $(1 - \delta)$ changes by less than a small amount (`tol`). Second, if the optimal action has not changed after a specified number of iterations (`nochangelim`). Finally `relval` and `vanish` apply to infinite horizon non-discounted problems and determines if and how Average Expected Reward problems are handled.

18.3 Fields for the results structure

- name: same as model.name field (helps link models and results)

- v: $n^s \times 1$ vector or (if `options.keepall=1`) $n^s \times T$ matrix of value functions or an n^{stage} cell array if $n^{stages} > 1$

- Ixopt: n^s vector of the values of $\mathcal{I}^x$ associated with the optimal strategy (may also be $n^s \times T$ or an n^{stage} cell array)

- Xopt: If the `model` variable contained the field `X` the optimal rows of X are extracted and placed in this field. This information can also be obtained using `getA`. Note that the `Xopt` field is not filled with models with stages.
- pstar: $n^s \times n^s$ transition matrix associated with the optimal infinite horizon strategy ($T = \infty$ only)
- algorithm: 'b', 'p' or 'f' for backwards iteration (finite horizon), policy iteration and function iteration. Also for non-discounted Average Reward problems the suffix 'vd' or 'rv' is appended to denote vanishing discount or relative value.
- iter: in infinite horizon models the number of iterations used
- change: in infinite horizon models the maximal change in the value function on the last iteration
- numnochange: in infinite horizon models the number of iterations since the last change in the strategy
- errors: a cell array containing error information
- warnings: a cell array containing warning information

The `results` variable or just the errors or warnings fields can be passed to `mdpreport` to display the meaning of the errors.

19 Additional Resources

Useful discussions of dynamic programming can be found in [2] and in [4]. Miranda and Fackler's text is associated with a set of MATLAB procedures (the CompEcon Toolbox). The examples in chapter 7 of their text can be run using MDPSOLVE with minor editing.

A good reference for applications of structured decision making to resource management problems see [5]. This text contains a useful discussion of adaptive management and partial observability.

References

[1] Robert G. Haight and Stephen Polasky. Optimal control of an invasive species with imperfect information about the level of infestation. *Resource and Energy Economics*, 32:519–533, 2010.

[2] Kenneth L. Judd. *Numerical Methods in Economics*. MIT Press, Cambridge, MA, 1998.

[3] William S. Lovejoy. Computationally feasible bounds for partially observed markov decision processes. *Operations Research*, 39:162–175, 1991.

[4] Mario J. Miranda and Paul L. Fackler. *Applied Computational Economics and Finance*. MIT Press, Cambridge MA, 2002.

[5] Byron K. Williams, James D. Nichols, and Michael J. Conroy. *Analysis and Management of Animal Populations*. Academic Press, San Diego, CA, 2001.

[6] Rong Zhou and Eric A. Hansen. An improved grid-based approximation algorithm for pomdps. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 707–716, 2001.

Index