MDPSolve enhancements in version 2014a.

Two major enhancements are included in the latest version of MDPSolve. A new model specification system is available that allows models to be specified in graphical form, similar to a Bayes Net or an Influence Diagram. With this feature models are specified by defining a set of variables along with their relationships. Variables are characterized by their type (states, actions, etc.), the values they can take on and their probability distributions, which can be dependent on other variables that have already been defined. A number of utilities are provided that will translate the graphical form into a form that MDPSolve can use to determine optimal strategies. A plotting procedure provides users with an interactive graphical representation of the system.

The second major enhancement allows the user to define models in an extended POMDP framework. This framework allows for a more general specification of the relationship of observable variables to system state variables. The generalization increases modelling flexibility and enhances the interpretation of results. The extended POMDP approach can be accessed using the graphical model specification system be defining some state variables as unobserved. The software will then determine how observed variables provide information that is useful for updating beliefs about unobserved variables. Adaptive management models can be specified by including unobserved parameters in the model.

Two additional enhancements help speed existing functions. First there is an alternative solution method for infinite horizon dynamic programming problems. In previous versions both function and policy iteration were available as well as Puterman's modified policy iteration. Policy iteration was implemented to use a direct approach to solving a system of linear equations. The latest version also provides for policy iteration based on an iterative (Krylov) linear equation solver rather than a direct method. This is often significantly faster and it does not require that the transition matrix be used directly. Instead the expected value function can be used and still obtain the benefits of using a policy iteration approach. For further discussion of this approach see Mico Mrkaic. Policy iteration accelerated with Krylov methods. Journal of Economic Dynamics & Control, 26 (2002) 517-545. This approach can be obtained by setting the `algorithm` option to 'i'. In future releases this option will become the default option for discounted infinite horizon problems.

The second speed enhancement is in computing category count transition matrices. When the individual site transition matrix is block diagonal (or can be reordered to be block diagonal) processing the individual blocks separately and then combining the results can result in large speed ups. If the individual site transition matrix is constant catcountP will automatically detect block diagonality and use the new approach. If the individual site transition matrix is specified as a function (because it depends on the number of sites in the various categories) a pattern matrix can be passed to catcountP to specify the block pattern of the individual site transition matrices. This pattern will be used to determine the block structure of the individual site transition matrices, which will then be used to compute the category count transition matrix.

Graphical Model Specification

The graphical model specification of a system defines a set of variables and the relationships among them. The system can be thought of as a formal graph with graph nodes representing variables and directed arcs or links representing the conditional probability relationships. Consider a system with two variables (nodes), A and B. If there is a directed link from A to B this implies that we can define the probability of B conditionally on A. In the language of graph theory A is a parent of B and B is child of A. Furthermore, the tail of the arc connecting A and B is attached to A and the head is attached to B.

Model specification can be thought of as consisting of two stages. In the first stage the graphical structure of the model can be defined. This consists of defining the variables and specifying the links between them. It can be useful to use interactive graphing software at this stage to help both analysts and clientele visualize the system being modeled. MDPSolve provides an interactive graphing procedure called `drawdiagram`. This tool can be used to plot either a prespecified model or to create one interactively from scratch. There is also a procedure that writes out MATLAB code to reconstruct a diagram that is created interactively.

The second stage of model specification fills in the details by defining the possible values each variable can take on and by defining the conditional probability distributions (CPDs) for (some of) the variables. MPDSolve provides three ways of defining relationships between a child variable and its parents. The basic method for defining this information uses an rv structure, which can be defined using the procedure `rvdef`. Variables can be defined as deterministic functions of their parent variables or as either discrete or continuous random variables. Discrete variables are defined by a Conditional Probability Table (CPT), which itself can be defined as a matrix or as a function of a set of parent variables. Continuous variables are defined for specific named distributions by specifying the parameters of the distribution, which, in turn, can depend on parent variables (currently this feature is not implemented; instead continuous variables with alternative parameters should be obtained using variable transformations). Currently MDPSolve has definitions for the Normal, Gamma, Beta, Burr-12, Kumaraswamy, linear and triangular distributions.

Once all of the variables in a diagram are specified the diagram must be processed into a form that MDPSolve can use to determine an optimal strategy. Essentially this means obtaining the reward function and the transition probability matrix (or its equivalent). This processing is carried out by the procedure `d2model`, which accepts a diagram and returns a model structure that can be passed to the `mdpsolve` procedure. There are a number of options that one can choose from to carry out this task. MDPSolve can construct the reward function and the transition probability matrix from the information supplied using either a discretization approach or a Monte Carlo (simulation) approach (using the discretization approach yields exact if all variables are, in fact, discrete). Furthermore `d2model` can construct a function that maps the future value function into the expected future value function conditional on the current state and action variables. In some cases this can be advantageous in lieu of constructing the entire transition matrix.

Programmatic Model Specification

The `add2diagram` procedure is the basic tool used to define a model programmatically. The arguments passed to this procedure are

diagram	an existing diagram	empty to start a new model
name	variable name	a string
type	variable type	‘s’, ‘a’, ‘f’, ‘u’, ‘c’ or ‘p’
obs	variable observed or not	true of false (1/0)
parents	names of parent variable	a cell array of strings
cpd	conditional probability distribution	a vector, function or rv structure

Model specification can be broken up into 2 steps. The first defines each variables name, type, observation status and parents. The second step defines the CPD information about the variable which includes information about whether it is discrete or continuous, about the possible values it can take on and about its conditional probability. Discussion of this second step will be deferred for now. If only 5 inputs are passed the CPD will be set to empty.

For example, suppose that there is a single state, which takes on values 0 and 1 and a single action that also takes on values of 0 and 1. Ignoring for now the CPD the following code defines the model.

```
D=[];
D=add2diagram(D,'State'  ,'s',true,{});
D=add2diagram(D,'Action' ,'a',true,{});
D=add2diagram(D,'State+' ,'f',true,{'State','Action'});
D=add2diagram(D,'Utility','u',true,{'State'});
```

Note that a variable with no parents is passed an empty cell array using curly brackets: {}. States and actions in general have no parents.

Using the `drawdiagram` procedure (`drawdiagram(D)`) yields the following visual representation of the model.

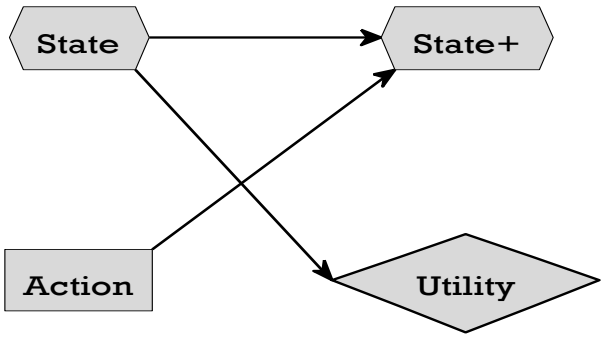

Figure 1. A simple model

Notice that different variable types have different shapes to make for easy identification. The Action, Utility and Chance types are standard shapes for influence diagrams. The State and Parameter shapes are

not part of standard Influence diagrams and have been given their own shapes. It is possible that the visual display of the figure can be improved by altering the location of the variables or their size. Interactive alterations of the diagram are discussed in the next section.

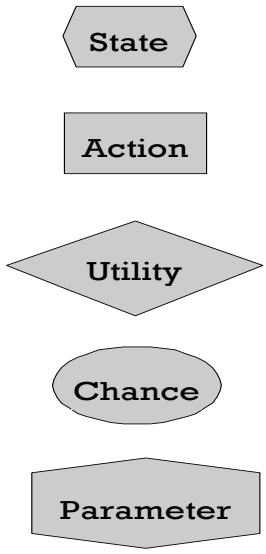

Figure 2. Shapes for the alternative variable types.

Interactive Model Specification

The `drawdiagram` procedure can be used interactively to specify the variables of a model and the parent/child relationships. To do so simply call `drawdiagram` without passing it a diagram structure. This opens a figure window. You can add variables by left clicking your mouse on any blank area in the figure window. A dialog box will open to allow you to specify the variables name (this must be specified), its type and whether it is observed or not. The variable name will appear at the point where you clicked.

Variables can be moved by left clicking on them and moving the mouse while holding the mouse button down. This allows you to arrange the variables to increase the readability of the plot.

Parent/child relationships can be specified by first right clicking on the parent variable. This will select the variable and its border will become red. Then right click on the child variable. You will be asked to confirm the change. If the relationship is not valid the request is rejected. In order to satisfy the rules of probability the graph must contain not cycles, that is, no paths should lead from a node back to itself. Any arc that would create a cycle is rejected. Other aspects of the graph are not currently checked however. In particular state and action variables should have no parents and the utility variable should have no children.

A variable can be removed from the plot by selecting the variable (right-clicking) and pressing the Delete key. When a variable is removed all arcs in and out of the variable are also removed. The characteristics of a variable can be altered by selecting the variable and pressing “e” (for edit). This will open the node specification dialog box to allow characteristics to be altered.

A variety of methods can be used to alter the appearance of the plot. PgUp and PgDown buttons increase or decrease the node size. Home and End expand or contract the plot from or to the middle of the figure window. The arrow keys shift the plot up, down, to the right or to the left. The figure window can also be resized in the usual way.

Several function keys perform various functions. F1 produces a help screen summarizing the operation of the plots. F2 and F3 allow you to change the node and background colors. F4 allows you to change the font name used for the variable labels. F5 toggles the figure windows menu and toolbar; this allows you, for example, allowing you to save and/or copy the figure.

The `updatediagram` procedure can be used to create a diagram structure or to print code to create or modify a diagram structure. As an example suppose that a diagram window is opened using

```
drawdiagram
```

After the nodes and arcs are defined use

```
D=updatediagram([],gcf,2);
```

This will return a diagram structure and print out code that can be cut and pasted into a script file to be rerun at a later time. The information printed to the screen will include two fields that can be added to the diagram structure that govern the appearance of the plotted diagram. D.locs contains information about the location of the nodes and D.arcs contains information about how arcs are attached to nodes.

The diagram that is created will not be complete because it will still lack the CPD data associated with each node. This process is discussed in the next section.

For example using updatediagram([],gcf,2) with the simple diagram shown above produces the code

```
D=add2diagram([],'State','s',true,{});
D=add2diagram(D,'Action','a',true,{});
D=add2diagram(D,'State+','f',true,{'State','Action'});
D=add2diagram(D,'Utility','u',true,{'State'});
D.locs=[ ...
0.332 0.332 0.665 0.665;
0.665 0.333 0.665 0.333]';
D.attachments=[ ...
 1  2  1;
 4  3  3;
 4  6  5;
 1  2  1]';
```

(each variable has 8 attachment points and the attachments information ensures that the arcs are attached at the desired attachment points). If `D` is plotted now (`drawdiagram(D)`) it will appear exactly as it was created interactively.

Specifying CPD information

As already mentioned all of the elements of the diagram can be specified interactively except the CPD information which is specified by defining a so-called rv (random variable) structure. This is done using the `rvdef` procedure. The basic syntax for the procedure is

```
rv=rvdef(type,parameters,values);
```

The first input is a string representing the type of random variable. Alternatives are listed in Table ??.

Table ??. RV types and associated information

	type	**parameters**	**values**	**other**
function	'f'	function	vector	
values only	'v'		vector	
discrete	'd'	matrix or function	vector	order
logit	'logit'	n-vector	n-vector	
normal	'n'	2-vector	vector	even: 0/1
gamma	'g'	2-vector	vector	
beta	'b'	2-vector	vector on [0,1]	
Kumaraswamy	'k'	2-vector	vector on [0,1]	
linear	'lin'	Scalar	vector on [0,1]	
triangular	'tri'	3-vector [a b c]	vector on [a,b]	

The values input, if specified, should be a column vector. This input may not be needed for chance ('c') type variables or for the utility ('u') variable. When the values vector is needed is discussed after the alternative methods of processing models is discussed.

For many problems the most natural way to specify a variable is to use a function. For simple functions MATLAB's anonymous function feature provides an intuitive approach but m-files functions can also be used. There are two main requirements in writing a functional representation of a variable. First, the input variables of the function must match the parent variables defined for the variable and be in the same order. Second, the function should be able to accept vector input arguments (all of the same length) and return a single vector output, also of the same length. This is best done using the element-by-element arithmetic operators such as `.*`, `./` and `.^`. As a shortcut if a function handle is passed to add2diagram an 'f' type rv structure will be created. Thus the following two lines produce identical results:

```
D=add2diagram(D,'Utility','u',true,{'State','Action'},Utility);
```

and

```
D=add2diagram(D,'Utility','u',true,{'State','Action'},rvdef('f',Utility));
```

where `Utility` is a function handle for a function of two inputs. In general values do not need to be specified for variables defined by functions unless one wants to define a CPT for the variable (or a discretized version of it). In general this is not necessary unless the variable is a future state variable. If the variable is a future state variable and has the same name as a current state variable with a '+' suffix then add2diagram will add the appropriate values vector when a function handle is passed. Thus the following set of code produce identical results

```
D=add2diagram([],'State', 's',true,{},S);
D=add2diagram(D, 'State+','f',true,{'State','Action'},Snext);
```

and

```
D=add2diagram([],'State', 's',true,{},[0;1]);
D=add2diagram(D, 'State+','f',true,{'State','Action'},rvdef('f',Snext,S));
```

Where `Snext` is a function handle for a function of two inputs.

Current state and action variables do not require conditional probability information as all conditioning is in terms of these variables. In general these can be specified as values only variables. The values specified might be the actual values if the variable is discrete or they could represent a discretization if the variable

is continuous. In either case a vector of values is needed to properly define a model. As a shortcut, a vector of values can be passed to add2diagram. Thus

```
D=add2diagram([],'State','s',true,{},[0;1]);
```

and

```
D=add2diagram([],'State','s',true,{},rvdef('v',[],[0;1]));
```

produce identical results.

Discrete Random Variables

Currently two types of discrete variables can be defined, 'd' and 'logit'. The former uses the syntax

```
rv=rvdef('d',parameters,values,order).
```

The parameters input is a CPT matrix of probability values or a function handle, values is a column vector containing the discrete set of values the variable can take on and order is a 2-character string describing how the CPT is organized. If values is omitted the default set of values `(1:n)'` is used, where *n* is the number of values, which must be determined from the CPT matrix (values must be specified if parameters is a function handle). If the parameters input is passed as a function handle it should return an *n*-row matrix of values, with each column representing the probability distribution of the variable conditioned on one combination of the parent variables.

When the parameters input is specified as a matrix it is important that you make sure that the ordering of the columns matches the ordering of the parent variables. Use `X=dvalues(D,parents);` to determine the size and ordering of the parent variables (this returns a cell array of vectors; for a matrix of values use `X=dvalues(D,parents,'m');`). Here `parents` is a cell array containing the names of the parents.

The example discussed above could be completed in the following way:

```
s=[0;1];
a=[1;2];
P=normalizeP(rand(2,4));
D=add2diagram([],'State',  's',1,{},                  s);
D=add2diagram(D, 'Action', 'a',1,{},                  a);
D=add2diagram(D, 'State+', 'f',1,{'State','Action'},rvdef('d',P,s));
D=add2diagram(D, 'Utility','u',1,'State',             @(S)S);
```

Here the future state is defined as a discrete variable with a 2x4 CPT P. We could also define the variable with a function that selects the appropriate columns of P. Here are two possibilities along with code that demonstrates that they produce the same results.

```
Pfunc1=@(S,A) P(:,gridmatch({S,A},{s,a}));
Pfunc2=@(S,A) P(:,A+2*S);
X=dvalues(D,{'State','Action'});
P, Pfunc1(X{:}), Pfunc2(X{:})
```

The first function used the MDPSolve `gridmatch` procedure to obtain the appropriate index values associated with the parent values S and A. The second one creates a custom function (A+2*S) that produces the same indices. The ability to use either a matrix or a function to define the conditional distribution gives you considerable flexibility.

In general it is assumed that a CPT is defined with rows associated with the variable and columns associated with the parent (conditioning) variables and that the values of the parents are in lexicographic order. This default assumption can be overridden by passed an `order` string to `rvdef`. The following table lists the alternative arrangements for the CPT matrix

`'lc'`	lexicographic/column stochastic
`'lr'`	lexicographic/row stochastic
`'rc'`	reverse lexicographic/column stochastic
`'rr'`	reverse lexicographic/row stochastic

When a CPT matrix is specified and simulation methods are used to process a model a function that maps parent values into an index of the columns of the CPT is required. The functions in the above example `gridmatch({S,A},{s,a})` and `A+2*S` perform this operation. When a CPT is specified such a function is automatically created when it is first needed. The function created is somewhat more efficient if the parent variables are all discrete and have values vectors that are composed of consecutive integer values, e.g., [0;1] or [1;2;3;4].

The logit type variable is specified using

```
rv=rvdef('logit',parameters,values);
```

The parameters input should be an *n*x*m* matrix where *m* is equal to 1 plus the number of parent variables and *n* is equal to the number of values. As above if values is omitted the default set of values `(1:n)'` is used. Let B represent the parameter matrix. Then the probability that the variable equals value *i* is proportional to $\exp\left(B_{i1} + \sum_{j=2}^{m} B_{ij}\, X_{j-1}\right)$ where X_j is the value of the *j*th parent variable. Note that logit parameters are often estimated with a reference category, generally either the first or last category. If this is the case a row of 0s should be concatenated to the top or bottom of the parameter matrix.

Continuous Random Variables

Continuous random variables are defined using the syntax rv=def(name,parameters,values), where name is a string indicating the desired family of probability distributions and parameters is a column vector or a function handle to a function of the parents that returns a matrix of parameter values with rows equal to the number of values of the parents. The last input, which is optional, is used to define a discretization of the random variable.

Parameters should be either a column vector of values or a function handle for a function of the parent variables that returns a matrix of values, with a column for each realization of the parent variables. The third input, values, can either be a positive integer n, in which case a standard discretization with n values is created. Alternatively it can be a cell array containing two column vectors with the values and CPT associated with the discretized variable. Note that values need be specified only if discretization methods are used.

One limitation imposed on modelling involves the use of continuous random variables with parameters that depend on parent variables. The limitation does not affect simulation methods but if discretization methods are used to obtain transition matrices or conditional expectation functions then the discretization of such a variable must result in a single vector of values that are themselves not functions of the parent variables (note that this is always true for discrete variables). In other words, the discretization method must actually result in a discrete random variable that replaces the original one.

This restriction generally arises when the parameters involve a change of location or scale such as with the normal or gamma distribution. It can also arise if the nodes used to discretize a distribution depend on the parameter values, such as the use of Gauss-Jacobi quadrature nodes and weights for the Beta distribution. In some cases there are simple workarounds. For example suppose y is normally distributed with parameters that depend on x (mu(x) and sigma(x)) and z is a function of y. Instead define y to be N(0,1) and make z depend on x and y using mu(x)+sigma(x)y to replace the original y. This is easy because the nodes for the normal distribution involve only a simple scale and location shift.

For Gamma and Beta variables it is not possible to use Gaussian quadrature nodes and weights as there is no simple closed form mapping between a standard set of nodes and the nodes for different parameter values. For the Gamma, if only the scale parameter depends on the parents then a standard discretization can be used and rescaled in child variables. For the Beta it is possible to use a standard set of nodes on [0,1], such as Gauss-Legendre nodes and still obtain reasonable results if the parameters always take on values greater than or equal to 1 and are not too big (parameter values below 1 lead to an infinite density at one or both endpoints which makes accurate integration difficult whereas large parameter values imply a density that is concentrated in a small part of the [0,1] interval).

Another workaround for any random variable uses the inverse CDF method. To implement this first define a random variable u that is uniformly distributed on [0,1] and then define y as a function of u using $y=F^{-1}(u;p(x))$, where F is the cumulative distribution function of y the parameters of which, p(x), can depend on another variable x. Inverse functions for the Gamma and Beta are available in Matlab (`gammaincinv` and `betaincinv`) and are used in MDPSolve to generate random variables for these distributions.

Processing a Model

Once a model is specified as a diagram it must be processed into a form that MDPSolve can use to solve with dynamic programming. There are two aspects to this process. One involves computing the reward function, which is the expectation of the utility variable conditioned on the current state and action variables. The second is to be able to compute the expectation of the future value function conditional on the current state and action variables. This must be done repeatedly by the dynamic programming algorithm and hence this tends to be the most critical operation. It also tends to be the most time consuming operation and hence should be performed as efficiently as possible.

There are two ways that the conditional expected value can be computed. One is to compute the transition probability matrix P and use E[V|X]=P'V. The other is to compute a function g such that E[V|X]= g(V), which can be done without ever computing P. In some cases this function can be computed more efficiently by utilizing the graphical structure of the model.

To compute the transition matrix two procedures are available. One uses Monte Carlo simulation to obtain the transition matrix. The other method provides an exact value so long as the system involves only discrete random variables. If some of the random variables are discretized then both methods are approximations. Furthermore, if the state variables are themselves discretizations of continuous variables then the transition matrix is a discrete approximation to the continuous probability distribution. Which method is best to use may require some experimentation.

All of the methods are most easily obtained using the `d2model` procedure, which converts a diagram into a model structure that can be passed to `mdpsolve`. The basic syntax is

```
model=d2model(D,options);
```
with the `options` structure controlling the algorithms used. The Monte Carlo approach is used if the field `reps` is set to a positive number representing how many paths (replications) are generated. This supersedes options associated with the other methods, which are ignored. The other option specific to the Monte Carlo approach is the chunk option. MATLAB tends to work faster when vectorized and chunk signals that groups of chunk state/action values should be processed simultaneously. Generally there is a chunk size that is optimal for a given system and some experimentation may be required to find it. Set it to a smaller number if out-of-memory problems arise.

The Monte Carlo approach works in the following way. For each of the possible values of the current state and action variables it simulates reps samples of the remaining variables over one time step. This produces reps values of the future state variables. The interpolation weights associated with each of these points relative to the discrete grid of values used in the DP representation are determined. These can be represented as a vector equal in size to the total number of state values. The mean of these interpolation weight vectors is used as an approximation to the conditional distribution of the future states. Care is taken to ensure that the same underlying random variables are used in the simulations for all of the values of the conditioning variables. This reduces the variability in the results and provides for consistency in the sense that the qualitative behavior of conditional expectation functions in the approximating model should match that of the true model.

If the `reps` option is set to 0 discretization methods are used to compute the reward and transition. The `ptype` option is used to control whether the transition matrix or the conditional expectation function is computed. Setting this option to 0 indicates that the conditional expectation approach should be used; otherwise the transition matrix is computed. To use either of these options any rv structure used to define a variable must be discretized. The simplest way to do this is to define the variable initially with a third argument passed to `rvdef` that controls the discretization. For example,
```
CPD=rvdef('n',[0 1],25);
```
indicates that a 25 value discretization should be used. Alternatively
```
CPD=rvdef('n',[0 1],{x,w});
```
can be used, with x and w representing a set of values and associated probability weights.

When the discretized model is treated as the true model and the exact transition matrix or conditional expectation function is requested, the basic underlying computational task is to solve a problem of the form

$$\sum_{i \in S} \prod_{i \in M} f_i(X_i)$$

where *M*, *S* and the X_i are subsets of the variables in the system with ($S \subseteq M$). Each factor f_i is a function of X_i and represents either a CPT or an array of values. Thus each factor is multi-dimensional array of numbers and the computational task is to perform a sum-product operation on the whole set of factors.

The sum product operation proceeds as a sequence of operations on pairs of factors, creating a new factor in the process. It is well known that the order of this sequence of operations makes a large difference in the efficiency with which the entire operation is performed. Unfortunately determining the optimal sequence is itself a difficult problems and, except with very small numbers of factors, approximate methods are used to determine the order of operations. In addition, standard methods to perform the binary operation (multidimensional multiplication and summation) do not seem to exist and there are many complicated considerations involved in performing this operation efficiently. MDPSolve utilizes

methods that seem to work reasonably well but more work is needed to ensure that these operations are done efficiently.

There are a number of choices that can be made to control the handling of the sum-product operation. There are three algorithms for determining the sequence order which are controlled by the option `orderalg`. The default method first performs all operations that do not result in an increase in the factor dimension (the number of variables involved). It then attempts to find an optimal sequence for the remaining set of factors. The second algorithm uses a greedy search approach that, in each step, combines the factors with the smallest resulting factor. The third approach does an exhaustive search to determine the optimal (this option can result in long processing times). Diagnostics on the behavior of the algorithms can be monitored by setting the `orderdisplay` option to 1 and the `print` option to 1 or 2.

The `d2model` procedure returns a model structure that can be passed to `mdpsolve` to obtain the optimal strategy. Before doing so, however, the discount factor field (d) and, for finite horizon models, the time horizon field (T) must be set. These can also be set in the options structure passed to `d2model`, as can the terminal value vector (vterm).

Use of Discrete Values of Variables

Discretized values of a continuous variable are used to generate a Conditional Probability Tables (CPT) for the variable if this is needed. Given that MDPSolve ultimately works with models in which the state, parameter and action variables are treated as discrete, all action and parameter variables as well as current and future states must be associated with a set of discrete values. The utility variable, on the other hand, generally does not need a set of associated discrete values unless it is actually a discrete variable. When the utility variable is continuous it is generally best to define it as a function of its parent variables.

For chance variables a discrete set of values is only needed when these variables are replaced by a CPT. This does not arise when Monte Carlo simulation methods are used to process a model. When discretization methods are used values must be specified for continuous random variables or for variables defined by functions when the `passforward` option is set to 0 for the variable when `d2model` is called.

Simulating a Model

A model specified as a diagram can be simulated using the `dsim` procedure. The syntax is

```
S=dsim(D,S0,rep,T,A,pval);
```

Where D is a diagram structure, S0 is a vector of initial state values, reps is the number of repetitions (samples) generated, T is the time horizon, A is a matrix mapping states into actions. Each row of this matrix given the values of the action variables at one of the grid values of the state variables. For states lying between grid points, the action associated with the nearest neighbor will be used. If there are parameter variables in the diagram a single instance of these variables should be passed as pval.

The procedure returns a cell array with an element for each variable in the diagram. Each element is a reps by T+1 matrix of values for the associated varaiable.

Code and Examples

Code for the diagrammatic approach is contained in the subdirectory influence and its subdirectories, One of these subdirectories, examples, contains a number of worked examples using the graphical specification approach.

Extended POMDP Models

MPDSolve now has the ability to solve the extended POMDP model discussed in Fackler and Pacifici, 2014. This framework allows any variable to be an observation variable. As the procedure is currently implemented one specifies such a model by providing the joint probability distribution of the future state and observation variables conditional on the current states and actions. The procedure `xpomdp` is called and returns model components with unobserved state variables and parameters replaced by a discretized belief distribution over these unobserved components. Note that parameters are treated as state variables that do not change over time (so future parameter values depend only on the associated current values and the transition matrix is an identity matrix).

There is also a link with the graphical approach. If one specifies a model with a diagram the procedure `d2model` will convert a model with unobserved states or actions into an extended POMDP model which can then be passed to `mdpsolve`. Currently to accomplish this `d2model` computes the joint distribution of future states, unobserved parameters and all observed chance variables. It then calls `xpomdp` to replace the unobserved states and parameters with discretized belief states. Parameters are treated as special because their values do not change and hence there is no need to specify both current and future values of these variables.

Note that adaptive management applications are models with unobserved parameters, which are currently assumed to take on a discrete set of values. The parameters could represent a discretization of a continuous parameter or could be model indicator variables (or a combination of these). Given the solution method used to solve these models (discretization of the belief states) the approach is limited to models with a relatively small number of values of the unobserved state and parameter values. Plans are underway for future versions of MDPSolve to provide methods other than discretization for handling unknown continuous state variables and parameters.

Advanced Features

Feasability and expansion. In many situations there are feasibility restrictions on actions such that not every action is possible for every value of the state variables. For example, if the state is the level of some resource and the action is the amount of the resource utilized then the action can be no bigger than the current state. MDPSolve is set up to handle situations in which the number of state/action combinations is less than the products of the number of state values and the number of action values. It does this by defining an index vector Ix which indicates which state value is associated with a given state/action combination (see the MDPSolve documentation for a fuller discussion of this principle).

The "standard" way of imposing feasibility constraints on a dynamic programming problem is to set the value of the reward function to a large negative number for any infeasible state/action combination. Doing this ensures that this state/action combination is so bad that it never gets picked. A simple way to do this when defining a reward (utility) function uses the `ifthenelse` procedure provided with the MDPsolve package. For example, suppose that the utility function is equal to the action divided by the state with the restriction that the action can be no larger than the state. One could use

```
U = @(S,A) ifthenelse(A<=S,A./(S+realmin),-inf);
```

to define the utility function. The utility function is given as A./(S+realmin), where adding the very small positive number realmin prevents division by 0, which would result in a value of NaN (not-a-number) and would cause numerical problems. The ifthenelse(c,a,b) function operates like the C operator c?a:b or the Basic function if(c,a,b), except that it can operate on arrays of values. For any value for which the condition c is true ifthenelse returns the associated value of a; for any false value of c it returns the associated value of b.

Another way to reduce the effective size of a problem (by reducing the amount of work performed) arises if the transition probabilities are the same for multiple state/action combinations. MDPSolve exploits this possibility when the model.Iexpand field is set. In general this occurs when the future state variables are not dependent on all of the state action variables. For example, if the state variable is the level of some resource and the action is the post-decision level of the resource, the future level of the resource may depend only on the current action and not on the current state (specifically, the future state is conditionally independent of the current state given the current action). When this possibility arises in a diagrammatic model it is automatically detected and an Iexpand vector is added to the model structure.

To illustrate these two features consider the problem of managing the harvest of a wild stock of some animal. The population dynamics are described by a function of the form $S^+=g(E)\exp(e)$, where S is the population level and E is the escapement level, which equals the population level minus the harvest level, and e is a normally distributed noise term. The following code sets up this model using a Beverton-Holt growth function with intrinsic growth rate r and normalized so the carrying capacity K equals 1. Notice that there are 100 values of the population level equally spaced on [0,1.5]. The escapament variable uses the same values. Thus there are 10,000 possible state/action combinations. Only 5,050 of these, however, are feasible.

The transition matrix for the feasible values is therefore 100x5050. Columns associated with the same escapement values, however, are all identical. It is therefore only necessary to compute the 100x100 matrix and then use the Ixpand vector to compute values for all 5050 feasible state/action combinations. Processing this model with d2model results in a model with a 100x100 transition matrix and 5050-element index vectors Ix and Ixpand.

```
r       = 0.1;                                  % intrinsic growth rate
K       = 1;                                    % carrying capacity
sigma   = 0.02;                                 % noise standard deviation
delta   = 0.98;                                 % discount factor
Snext   = @(E,z) (1+r)*E./(1+(r/K)*E).*exp(z); % next period state
Utility = @(S,E) ifthenelse(E<=S,S-E,-inf);     % utility function
Smax    = 1.5*K;                                % maximum state value
ns      = 100;                                  % number of state values
ne      = 10;                                   % number of noise values
S       = linspace(0,Smax,ns)';                 % population values
pe      = rvdef('ne',[-sigma^2/2;sigma],ne);    % noise distribution
D=add2diagram([],'population', 's',1,{}                          ,S);
D=add2diagram(D ,'escapement', 'a',1,{}                          ,S);
D=add2diagram(D ,'noise',      'c',1,{}                          ,pe);
D=add2diagram(D ,'population+','f',1,{'escapement','noise'}      ,Snext);
D=add2diagram(D ,'Utility',    'r',1,{'population','escapement'},Utility);
```

The diagrammatic representation of the model is shown in Figure ??. Notice that, given the current action (escapement), the future state (population+) does not depend (either directly or indirectly) on the current state.

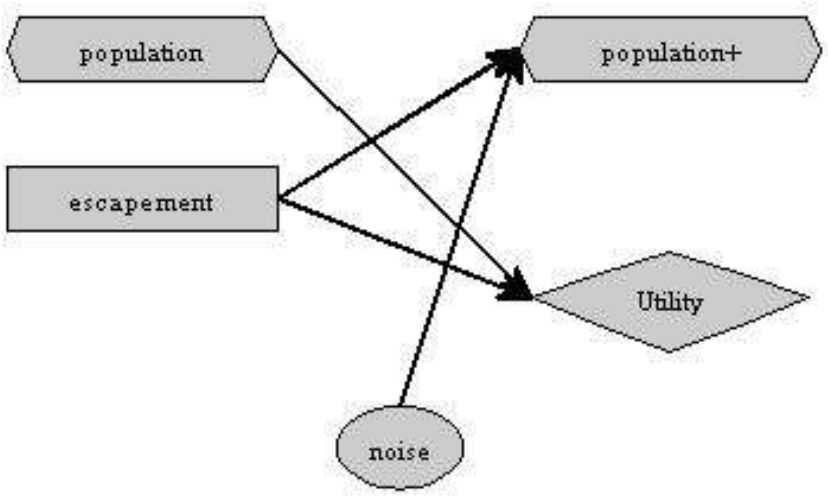

Figure ??. Diagrammatic Representation of the Simple Harvest Model

Correlated Noise

In many situation correlated noise terms arise. When the noise terms are individually normal, a set of correlated noise terms can be obtained by defining an underlying set of independent noise variables and then create a new set using linear combinations of the original. To facilitate this process the utility `addcorrrnoise` is provided. To use this procedure first define an *m*x*m* correlation matrix C and add this to an existing diagram using D=addcorrnoise(D,C); This will add *m* variables names z1,..,zm representing the underlying independent standard normal and *m* variables named e1,…,em representing the correlated standard normal variables.

Defining new rv types

To define a new discrete type rv you must create two functions one with suffix 'def' and one with suffix 'gen' which describe how to define and simulate (generate) the rv. If the rv is continuous you should also define a function with suffix 'nw' that defines how the rv is discretized. [TO BE CONTINUED]