Python Markov Decision Process Toolbox Documentation

Release 4.0-b4

Steven A W Cordwell

April 13, 2015

Contents

The MDP toolbox provides classes and functions for the resolution of descrete-time Markov Decision Processes. The list of algorithms that have been implemented includes backwards induction, linear programming, policy iteration, q-learning and value iteration along with several variations.

The classes and functions were developped based on the MATLAB MDP toolbox by the Biometry and Artificial Intelligence Unit of INRA Toulouse (France). There are editions available for MATLAB, GNU Octave, Scilab and R. The suite of MDP toolboxes are described in Chades I, Chapron G, Cros M-J, Garcia F & Sabbadin R (2014) 'MDPtoolbox: a multi-platform toolbox to solve stochastic dynamic programming problems', *Ecography*, vol. 37, no. 9, pp. 916–920, doi 10.1111/ecog.00888.

CHAPTER 1

Features

- Eight MDP algorithms implemented
- Fast array manipulation using NumPy
- Full sparse matrix support using SciPy's sparse package
- Optional linear programming support using cvxopt

PLEASE NOTE: the linear programming algorithm is currently unavailable except for testing purposes due to incorrect behaviour.

CHAPTER 2

Installation

NumPy and SciPy must be on your system to use this toolbox. Please have a look at their documentation to get them installed. If you are installing onto Ubuntu or Debian and using Python 2 then this will pull in all the dependencies:

```
sudo apt-get install python-numpy python-scipy python-cvxopt
```

On the other hand, if you are using Python 3 then cvxopt will have to be compiled (pip will do it automatically). To get NumPy, SciPy and all the dependencies to have a fully featured cvxopt then run:

```
sudo apt-get install python3-numpy python3-scipy liblapack-dev
libatlas-base-dev libgsl0-dev fftw-dev libglpk-dev libdsdp-dev
```

The two main ways of downloading the package is either from the Python Package Index or from GitHub. Both of these are explained below.

2.1 Python Package Index (PyPI)

The toolbox's PyPI page is https://pypi.python.org/pypi/pymdptoolbox/ and there are both zip and tar.gz archive options available that can be downloaded. However, I recommend using pip to install the toolbox if you have it available. Just type

```
pip install pymdptoolbox
```

at the console and it should take care of downloading and installing everything for you. If you also want cvxopt to be automatically downloaded and installed so that you can help test the linear programming algorithm then type

```
pip install "pymdptoolbox[LP]"
```

If you want it to be installed just for you rather than system wide then do

```
pip install --user pymdptoolbox
```

If you downloaded the package manually from PyPI

1. Extract the **.zip* or **.tar.gz* archive

   ```
   tar -xzvf pymdptoolbox-<VERSION>.tar.gz
   ```

   ```
   unzip pymdptoolbox-<VERSION>
   ```

2. Change to the PyMDPtoolbox directory

   ```
   cd pymdptoolbox
   ```

3. Install via Setuptools, either to the root filesystem or to your home directory if you don't have administrative access.

```
python setup.py install
```

```
python setup.py install --user
```

Read the Setuptools documentation for more advanced information.

Of course you can also use virtualenv or simply just unpack it to your working directory.

2.2 GitHub

Clone the Git repository

```
git clone https://github.com/sawcordwell/pymdptoolbox.git
```

and then follow from step two above. To learn how to use Git then I reccomend reading the freely available Pro Git book written by Scott Chacon and Ben Straub and published by Apress.

CHAPTER 3

Quick Use

Start Python in your favourite way. The following example shows you how to import the module, set up an example Markov decision problem using a discount value of 0.9, solve it using the value iteration algorithm, and then check the optimal policy.

```
import mdptoolbox.example
P, R = mdptoolbox.example.forest()
vi = mdptoolbox.mdp.ValueIteration(P, R, 0.9)
vi.run()
vi.policy # result is (0, 0, 0)
```

CHAPTER 4

Documentation

Documentation is available at http://pymdptoolbox.readthedocs.org/ and also as docstrings in the module code. If you use IPython to work with the toolbox, then you can view the docstrings by using a question mark `?`. For example:

```
import mdptoolbox
mdptoolbox?<ENTER>
mdptoolbox.mdp?<ENTER>
mdptoolbox.mdp.ValueIteration?<ENTER>
```

will display the relevant documentation.

CHAPTER 5

Contribute

Issue Tracker: https://github.com/sawcordwell/pymdptoolbox/issues

Source Code: https://github.com/sawcordwell/pymdptoolbox

CHAPTER 6

Support

Use the issue tracker.

CHAPTER 7

License

The project is licensed under the BSD license. See LICENSE.txt for details.

CHAPTER 8

Contents

8.1 Markov Decision Process (MDP) Toolbox

The MDP toolbox provides classes and functions for the resolution of descrete-time Markov Decision Processes.

8.1.1 Available modules

example Examples of transition and reward matrices that form valid MDPs

mdp Makov decision process algorithms

util Functions for validating and working with an MDP

8.1.2 How to use the documentation

Documentation is available both as docstrings provided with the code and in html or pdf format from The MDP toolbox homepage. The docstring examples assume that the `mdptoolbox` package is imported like so:

```
>>> import mdptoolbox
```

To use the built-in examples, then the `example` module must be imported:

```
>>> import mdptoolbox.example
```

Once the `example` module has been imported, then it is no longer neccesary to issue `import mdptoolbox`.

Code snippets are indicated by three greater-than signs:

```
>>> x = 17
>>> x = x + 1
>>> x
18
```

The documentation can be displayed with IPython. For example, to view the docstring of the ValueIteration class use `mdp.ValueIteration?<ENTER>`, and to view its source code use `mdp.ValueIteration??<ENTER>`.

8.1.3 Acknowledgments

This module is modified from the MDPtoolbox (c) 2009 INRA available at http://www.inra.fr/mia/T/MDPtoolbox/.

8.2 Markov Decision Process (MDP) Toolbox: mdp module

The mdp module provides classes for the resolution of descrete-time Markov Decision Processes.

8.2.1 Available classes

MDP Base Markov decision process class

FiniteHorizon Backwards induction finite horizon MDP

PolicyIteration Policy iteration MDP

PolicyIterationModified Modified policy iteration MDP

QLearning Q-learning MDP

RelativeValueIteration Relative value iteration MDP

ValueIteration Value iteration MDP

ValueIterationGS Gauss-Seidel value iteration MDP

class mdptoolbox.mdp.**FiniteHorizon**(*transitions*, *reward*, *discount*, *N*, *h=None*, *skip_check=False*)
Bases: mdptoolbox.mdp.MDP

A MDP solved using the finite-horizon backwards induction algorithm.

Parameters

- **transitions** (*array*) – Transition probability matrices. See the documentation for the MDP class for details.
- **reward** (*array*) – Reward matrices or vectors. See the documentation for the MDP class for details.
- **discount** (*float*) – Discount factor. See the documentation for the MDP class for details.
- **N** (*int*) – Number of periods. Must be greater than 0.
- **h** (*array, optional*) – Terminal reward. Default: a vector of zeros.
- **skip_check** (*bool*) – By default we run a check on the transitions and rewards arguments to make sure they describe a valid MDP. You can set this argument to True in order to skip this check.
- **Attributes** (*Data*) –
- **————————** –
- **V** (*array*) – Optimal value function. Shape = (S, N+1). V[:, n] = optimal value function at stage n with stage in {0, 1...N-1}. V[:, N] value function for terminal stage.
- **policy** (*array*) – Optimal policy. policy[:, n] = optimal policy at stage n with stage in {0, 1...N}. policy[:, N] = policy for stage N.
- **time** (*float*) – used CPU time

Notes

In verbose mode, displays the current stage and policy transpose.

Examples

```
>>> import mdptoolbox, mdptoolbox.example
>>> P, R = mdptoolbox.example.forest()
>>> fh = mdptoolbox.mdp.FiniteHorizon(P, R, 0.9, 3)
>>> fh.run()
>>> fh.V
array([[ 2.6973,  0.81  ,  0.    ,  0.    ],
       [ 5.9373,  3.24  ,  1.    ,  0.    ],
       [ 9.9373,  7.24  ,  4.    ,  0.    ]])
>>> fh.policy
array([[0, 0, 0],
       [0, 0, 1],
       [0, 0, 0]])
```

run()

setSilent()
Set the MDP algorithm to silent mode.

setVerbose()
Set the MDP algorithm to verbose mode.

class mdptoolbox.mdp.**MDP**(*transitions*, *reward*, *discount*, *epsilon*, *max_iter*, *skip_check=False*)
Bases: builtins.object

A Markov Decision Problem.

Let S = the number of states, and A = the number of acions.

Parameters

- **transitions** (*array*) – Transition probability matrices. These can be defined in a variety of ways. The simplest is a numpy array that has the shape (A, S, S), though there are other possibilities. It can be a tuple or list or numpy object array of length A, where each element contains a numpy array or matrix that has the shape (S, S). This "list of matrices" form is useful when the transition matrices are sparse as scipy.sparse.csr_matrix matrices can be used. In summary, each action's transition matrix must be indexable like transitions[a] where a {0, 1...A-1}, and transitions[a] returns an S × S array-like object.
- **reward** (*array*) – Reward matrices or vectors. Like the transition matrices, these can also be defined in a variety of ways. Again the simplest is a numpy array that has the shape (S, A), (S,) or (A, S, S). A list of lists can be used, where each inner list has length S and the outer list has length A. A list of numpy arrays is possible where each inner array can be of the shape (S,), (S, 1), (1, S) or (S, S). Also scipy.sparse.csr_matrix can be used instead of numpy arrays. In addition, the outer list can be replaced by any object that can be indexed like reward[a] such as a tuple or numpy object array of length A.
- **discount** (*float*) – Discount factor. The per time-step discount factor on future rewards. Valid values are greater than 0 upto and including 1. If the discount factor is 1, then convergence is cannot be assumed and a warning will be displayed. Subclasses of MDP may pass None in the case where the algorithm does not use a discount factor.
- **epsilon** (*float*) – Stopping criterion. The maximum change in the value function at each iteration is compared against epsilon. Once the change falls below this value, then the value function is considered to have converged to the optimal value function. Subclasses of MDP may pass None in the case where the algorithm does not use an epsilon-optimal stopping criterion.

- **max_iter** (*int*) – Maximum number of iterations. The algorithm will be terminated once this many iterations have elapsed. This must be greater than 0 if specified. Subclasses of `MDP` may pass `None` in the case where the algorithm does not use a maximum number of iterations.
- **skip_check** (*bool*) – By default we run a check on the `transitions` and `rewards` arguments to make sure they describe a valid MDP. You can set this argument to True in order to skip this check.

P
array

Transition probability matrices.

R
array

Reward vectors.

V
tuple

The optimal value function. Each element is a float corresponding to the expected value of being in that state assuming the optimal policy is followed.

discount
float

The discount rate on future rewards.

max_iter
int

The maximum number of iterations.

policy
tuple

The optimal policy.

time
float

The time used to converge to the optimal policy.

verbose
boolean

Whether verbose output should be displayed or not.

run()
Implemented in child classes as the main algorithm loop. Raises an exception if it has not been overridden.

setSilent()
Turn the verbosity off

setVerbose()
Turn the verbosity on

run()
Raises error because child classes should implement this function.

setSilent()
Set the MDP algorithm to silent mode.

setVerbose()
Set the MDP algorithm to verbose mode.

class mdptoolbox.mdp.**PolicyIteration**(*transitions*, *reward*, *discount*, *policy0=None*, *max_iter=1000*, *eval_type=0*, *skip_check=False*)

Bases: mdptoolbox.mdp.MDP

A discounted MDP solved using the policy iteration algorithm.

Parameters

- **transitions** (*array*) – Transition probability matrices. See the documentation for the MDP class for details.
- **reward** (*array*) – Reward matrices or vectors. See the documentation for the MDP class for details.
- **discount** (*float*) – Discount factor. See the documentation for the MDP class for details.
- **policy0** (*array, optional*) – Starting policy.
- **max_iter** (*int, optional*) – Maximum number of iterations. See the documentation for the MDP class for details. Default is 1000.
- **eval_type** (*int or string, optional*) – Type of function used to evaluate policy. 0 or "matrix" to solve as a set of linear equations. 1 or "iterative" to solve iteratively. Default: 0.
- **skip_check** (*bool*) – By default we run a check on the transitions and rewards arguments to make sure they describe a valid MDP. You can set this argument to True in order to skip this check.
- **Attributes** (*Data*) –
- **————————** –
- **V** (*tuple*) – value function
- **policy** (*tuple*) – optimal policy
- **iter** (*int*) – number of done iterations
- **time** (*float*) – used CPU time

Notes

In verbose mode, at each iteration, displays the number of differents actions between policy n-1 and n

Examples

```
>>> import mdptoolbox, mdptoolbox.example
>>> P, R = mdptoolbox.example.rand(10, 3)
>>> pi = mdptoolbox.mdp.PolicyIteration(P, R, 0.9)
>>> pi.run()

>>> P, R = mdptoolbox.example.forest()
>>> pi = mdptoolbox.mdp.PolicyIteration(P, R, 0.9)
>>> pi.run()
>>> expected = (26.244000000000014, 29.484000000000016, 33.484000000000016)
>>> all(expected[k] - pi.V[k] < 1e-12 for k in range(len(expected)))
True
```

```
>>> pi.policy
(0, 0, 0)
```

run()

setSilent()
Set the MDP algorithm to silent mode.

setVerbose()
Set the MDP algorithm to verbose mode.

class mdptoolbox.mdp.**PolicyIterationModified**(*transitions*, *reward*, *discount*, *epsilon=0.01*, *max_iter=10*, *skip_check=False*)

Bases: mdptoolbox.mdp.PolicyIteration

A discounted MDP solved using a modifified policy iteration algorithm.

Parameters

- **transitions** (*array*) – Transition probability matrices. See the documentation for the MDP class for details.
- **reward** (*array*) – Reward matrices or vectors. See the documentation for the MDP class for details.
- **discount** (*float*) – Discount factor. See the documentation for the MDP class for details.
- **epsilon** (*float, optional*) – Stopping criterion. See the documentation for the MDP class for details. Default: 0.01.
- **max_iter** (*int, optional*) – Maximum number of iterations. See the documentation for the MDP class for details. Default is 10.
- **skip_check** (*bool*) – By default we run a check on the `transitions` and `rewards` arguments to make sure they describe a valid MDP. You can set this argument to True in order to skip this check.
- **Attributes** (*Data*) –
- **————————** –
- **V** (*tuple*) – value function
- **policy** (*tuple*) – optimal policy
- **iter** (*int*) – number of done iterations
- **time** (*float*) – used CPU time

Examples

```
>>> import mdptoolbox, mdptoolbox.example
>>> P, R = mdptoolbox.example.forest()
>>> pim = mdptoolbox.mdp.PolicyIterationModified(P, R, 0.9)
>>> pim.run()
>>> pim.policy
(0, 0, 0)
>>> expected = (21.81408652334702, 25.054086523347017, 29.054086523347017)
>>> all(expected[k] - pim.V[k] < 1e-12 for k in range(len(expected)))
True
```

run()

setSilent()
Set the MDP algorithm to silent mode.

setVerbose()
Set the MDP algorithm to verbose mode.

class mdptoolbox.mdp.**QLearning**(*transitions*, *reward*, *discount*, *n_iter=10000*, *skip_check=False*)
Bases: mdptoolbox.mdp.MDP

A discounted MDP solved using the Q learning algorithm.

Parameters

- **transitions** (*array*) – Transition probability matrices. See the documentation for the MDP class for details.
- **reward** (*array*) – Reward matrices or vectors. See the documentation for the MDP class for details.
- **discount** (*float*) – Discount factor. See the documentation for the MDP class for details.
- **n_iter** (*int, optional*) – Number of iterations to execute. This is ignored unless it is an integer greater than the default value. Defaut: 10,000.
- **skip_check** (*bool*) – By default we run a check on the transitions and rewards arguments to make sure they describe a valid MDP. You can set this argument to True in order to skip this check.
- **Attributes** (*Data*) –
- **————————** –
- **Q** (*array*) – learned Q matrix (SxA)
- **V** (*tuple*) – learned value function (S).
- **policy** (*tuple*) – learned optimal policy (S).
- **mean_discrepancy** (*array*) – Vector of V discrepancy mean over 100 iterations. Then the length of this vector for the default value of N is 100 (N/100).
- **Examples** –
- **———** –
- **# These examples are reproducible only if random seed is set to 0 in** (>>>) –
- **# both the random and numpy.random modules.** (>>>) –
- **import numpy as np** (>>>) –
- **import mdptoolbox, mdptoolbox.example** (>>>) –
- **np.random.seed(0)** (>>>) –
- **P, R = mdptoolbox.example.forest()** (>>>) –
- **ql = mdptoolbox.mdp.QLearning(P, R, 0.96)** (>>>) –
- **ql.run()** (>>>) –
- **ql.Q** (>>>) –
- **11.198909 , 10.34652034],** (*array([[]* – [10.74229967, 11.74105792], [2.86980001, 12.25973286]])
- **expected = (11.198908998901134, 11.741057920409865, 12.259732864170232)** (>>>) –

- **all(expected[k] - ql.V[k] < 1e-12 for k in range(len(expected)))** (>>>) –
- **True** –
- **ql.policy** (>>>) –
- **1, 1)** (*(0,)* –
- **import mdptoolbox** (>>>) –
- **import numpy as np** –
- **P = np.array([[[0.5, 0.5],[0.8, 0.2]],[[0, 1],[0.1, 0.9]]])** (>>>) –
- **R = np.array([[5, 10], [-1, 2]])** (>>>) –
- **np.random.seed(0)** –
- **ql = mdptoolbox.mdp.QLearning(P, R, 0.9)** (>>>) –
- **ql.run()** –
- **ql.Q** –
- **33.33010866, 40.82109565],** (*array([]*) – [34.37431041, 29.67236845]])
- **expected = (40.82109564847122, 34.37431040682546)** (>>>) –
- **all(expected[k] - ql.V[k] < 1e-12 for k in range(len(expected)))** –
- **True** –
- **ql.policy** –
- **0)** (*(1,)* –

run()

setSilent()
Set the MDP algorithm to silent mode.

setVerbose()
Set the MDP algorithm to verbose mode.

class mdptoolbox.mdp.**RelativeValueIteration**(*transitions*, *reward*, *epsilon=0.01*, *max_iter=1000*, *skip_check=False*)

Bases: mdptoolbox.mdp.MDP

A MDP solved using the relative value iteration algorithm.

Parameters

- **transitions** (*array*) – Transition probability matrices. See the documentation for the MDP class for details.
- **reward** (*array*) – Reward matrices or vectors. See the documentation for the MDP class for details.
- **epsilon** (*float, optional*) – Stopping criterion. See the documentation for the MDP class for details. Default: 0.01.
- **max_iter** (*int, optional*) – Maximum number of iterations. See the documentation for the MDP class for details. Default: 1000.
- **skip_check** (*bool*) – By default we run a check on the transitions and rewards arguments to make sure they describe a valid MDP. You can set this argument to True in order to skip this check.
- **Attributes** (*Data*) –

- ———————— –
- **policy** (*tuple*) – epsilon-optimal policy
- **average_reward** (*tuple*) – average reward of the optimal policy
- **cpu_time** (*float*) – used CPU time

Notes

In verbose mode, at each iteration, displays the span of U variation and the condition which stopped iterations : epsilon-optimum policy found or maximum number of iterations reached.

Examples

```
>>> import mdptoolbox, mdptoolbox.example
>>> P, R = mdptoolbox.example.forest()
>>> rvi = mdptoolbox.mdp.RelativeValueIteration(P, R)
>>> rvi.run()
>>> rvi.average_reward
3.2399999999999993
>>> rvi.policy
(0, 0, 0)
>>> rvi.iter
4

>>> import mdptoolbox
>>> import numpy as np
>>> P = np.array([[[0.5, 0.5],[0.8, 0.2]],[[0, 1],[0.1, 0.9]]])
>>> R = np.array([[5, 10], [-1, 2]])
>>> rvi = mdptoolbox.mdp.RelativeValueIteration(P, R)
>>> rvi.run()
>>> expected = (10.0, 3.885235246411831)
>>> all(expected[k] - rvi.V[k] < 1e-12 for k in range(len(expected)))
True
>>> rvi.average_reward
3.8852352464118312
>>> rvi.policy
(1, 0)
>>> rvi.iter
29
```

run()

setSilent()
Set the MDP algorithm to silent mode.

setVerbose()
Set the MDP algorithm to verbose mode.

class mdptoolbox.mdp.**ValueIteration**(*transitions*, *reward*, *discount*, *epsilon=0.01*, *max_iter=1000*, *initial_value=0*, *skip_check=False*)

Bases: mdptoolbox.mdp.MDP

A discounted MDP solved using the value iteration algorithm.

ValueIteration applies the value iteration algorithm to solve a discounted MDP. The algorithm consists of solving Bellman's equation iteratively. Iteration is stopped when an epsilon-optimal policy is found or after a specified number (max_iter) of iterations. This function uses verbose and silent modes. In verbose mode, the function

displays the variation of V (the value function) for each iteration and the condition which stopped the iteration: epsilon-policy found or maximum number of iterations reached.

Parameters

- **transitions** (*array*) – Transition probability matrices. See the documentation for the MDP class for details.
- **reward** (*array*) – Reward matrices or vectors. See the documentation for the MDP class for details.
- **discount** (*float*) – Discount factor. See the documentation for the MDP class for details.
- **epsilon** (*float, optional*) – Stopping criterion. See the documentation for the MDP class for details. Default: 0.01.
- **max_iter** (*int, optional*) – Maximum number of iterations. If the value given is greater than a computed bound, a warning informs that the computed bound will be used instead. By default, if discount is not equal to 1, a bound for max_iter is computed, otherwise max_iter = 1000. See the documentation for the MDP class for further details.
- **initial_value** (*array, optional*) – The starting value function. Default: a vector of zeros.
- **skip_check** (*bool*) – By default we run a check on the transitions and rewards arguments to make sure they describe a valid MDP. You can set this argument to True in order to skip this check.
- **Attributes** (*Data*) –
- **————————** –
- **V** (*tuple*) – The optimal value function.
- **policy** (*tuple*) – The optimal policy function. Each element is an integer corresponding to an action which maximises the value function in that state.
- **iter** (*int*) – The number of iterations taken to complete the computation.
- **time** (*float*) – The amount of CPU time used to run the algorithm.

run()
Do the algorithm iteration.

setSilent()
Sets the instance to silent mode.

setVerbose()
Sets the instance to verbose mode.

Notes

In verbose mode, at each iteration, displays the variation of V and the condition which stopped iterations: epsilon-optimum policy found or maximum number of iterations reached.

Examples

```
>>> import mdptoolbox, mdptoolbox.example
>>> P, R = mdptoolbox.example.forest()
>>> vi = mdptoolbox.mdp.ValueIteration(P, R, 0.96)
>>> vi.verbose
False
```

```
>>> vi.run()
>>> expected = (5.93215488, 9.38815488, 13.38815488)
>>> all(expected[k] - vi.V[k] < 1e-12 for k in range(len(expected)))
True
>>> vi.policy
(0, 0, 0)
>>> vi.iter
4

>>> import mdptoolbox
>>> import numpy as np
>>> P = np.array([[[0.5, 0.5],[0.8, 0.2]],[[0, 1],[0.1, 0.9]]])
>>> R = np.array([[5, 10], [-1, 2]])
>>> vi = mdptoolbox.mdp.ValueIteration(P, R, 0.9)
>>> vi.run()
>>> expected = (40.048625392716815, 33.65371175967546)
>>> all(expected[k] - vi.V[k] < 1e-12 for k in range(len(expected)))
True
>>> vi.policy
(1, 0)
>>> vi.iter
26

>>> import mdptoolbox
>>> import numpy as np
>>> from scipy.sparse import csr_matrix as sparse
>>> P = [None] * 2
>>> P[0] = sparse([[0.5, 0.5],[0.8, 0.2]])
>>> P[1] = sparse([[0, 1],[0.1, 0.9]])
>>> R = np.array([[5, 10], [-1, 2]])
>>> vi = mdptoolbox.mdp.ValueIteration(P, R, 0.9)
>>> vi.run()
>>> expected = (40.048625392716815, 33.65371175967546)
>>> all(expected[k] - vi.V[k] < 1e-12 for k in range(len(expected)))
True
>>> vi.policy
(1, 0)
```

run()

setSilent()
Set the MDP algorithm to silent mode.

setVerbose()
Set the MDP algorithm to verbose mode.

class mdptoolbox.mdp.**ValueIterationGS**(*transitions*, *reward*, *discount*, *epsilon=0.01*, *max_iter=10*, *initial_value=0*, *skip_check=False*)

Bases: mdptoolbox.mdp.ValueIteration

A discounted MDP solved using the value iteration Gauss-Seidel algorithm.

Parameters

- **transitions** (*array*) – Transition probability matrices. See the documentation for the MDP class for details.
- **reward** (*array*) – Reward matrices or vectors. See the documentation for the MDP class for details.
- **discount** (*float*) – Discount factor. See the documentation for the MDP class for details.

- **epsilon** (*float, optional*) – Stopping criterion. See the documentation for the `MDP` class for details. Default: 0.01.
- **max_iter** (*int, optional*) – Maximum number of iterations. See the documentation for the `MDP` and `ValueIteration` classes for details. Default: computed.
- **initial_value** (*array, optional*) – The starting value function. Default: a vector of zeros.
- **skip_check** (*bool*) – By default we run a check on the `transitions` and `rewards` arguments to make sure they describe a valid MDP. You can set this argument to True in order to skip this check.
- **Attribues** (*Data*) –
- **————** –
- **policy** (*tuple*) – epsilon-optimal policy
- **iter** (*int*) – number of done iterations
- **time** (*float*) – used CPU time

Notes

In verbose mode, at each iteration, displays the variation of V and the condition which stopped iterations: epsilon-optimum policy found or maximum number of iterations reached.

Examples

```
>>> import mdptoolbox.example, numpy as np
>>> P, R = mdptoolbox.example.forest()
>>> vigs = mdptoolbox.mdp.ValueIterationGS(P, R, 0.9)
>>> vigs.run()
>>> expected = (25.5833879767579, 28.830654635546928, 32.83065463554693)
>>> all(expected[k] - vigs.V[k] < 1e-12 for k in range(len(expected)))
True
>>> vigs.policy
(0, 0, 0)
```

run()

setSilent()
Set the MDP algorithm to silent mode.

setVerbose()
Set the MDP algorithm to verbose mode.

8.3 Markov Decision Process (MDP) Toolbox: `util` module

The `util` module provides functions to check that an MDP is validly described. There are also functions for working with MDPs while they are being solved.

8.3.1 Available functions

check() Check that an MDP is properly defined

checkSquareStochastic() Check that a matrix is square and stochastic

getSpan() Calculate the span of an array

isNonNegative() Check if a matrix has only non-negative elements

isSquare() Check if a matrix is square

isStochastic() Check if a matrix is row stochastic

mdptoolbox.util.**check**(*P*, *R*)

Check if `P` and `R` define a valid Markov Decision Process (MDP).

Let `S` = number of states, `A` = number of actions.

Parameters

- **P** (*array*) – The transition matrices. It can be a three dimensional array with a shape of (A, S, S). It can also be a one dimensional arraye with a shape of (A,), where each element contains a matrix of shape (S, S) which can possibly be sparse.
- **R** (*array*) – The reward matrix. It can be a three dimensional array with a shape of (S, A, A). It can also be a one dimensional array with a shape of (A,), where each element contains matrix with a shape of (S, S) which can possibly be sparse. It can also be an array with a shape of (S, A) which can possibly be sparse.

Notes

Raises an error if `P` and `R` do not define a MDP.

Examples

```
>>> import mdptoolbox, mdptoolbox.example
>>> P_valid, R_valid = mdptoolbox.example.rand(100, 5)
>>> mdptoolbox.util.check(P_valid, R_valid) # Nothing should happen
>>>
>>> import numpy as np
>>> P_invalid = np.random.rand(5, 100, 100)
>>> mdptoolbox.util.check(P_invalid, R_valid) # Raises an exception
Traceback (most recent call last):
...
StochasticError:...
```

mdptoolbox.util.**checkSquareStochastic**(*matrix*)

Check if `matrix` is a square and row-stochastic.

To pass the check the following conditions must be met:

•The matrix should be square, so the number of columns equals the number of rows.

•The matrix should be row-stochastic so the rows should sum to one.

•Each value in the matrix must be positive.

If the check does not pass then a mdptoolbox.util.Invalid

Parameters **matrix** (*numpy.ndarray, scipy.sparse.*_matrix*) – A two dimensional array (matrix).

Notes

Returns None if no error has been detected, else it raises an error.

mdptoolbox.util.**getSpan**(*array*)
Return the span of *array*

span(array) = max array(s) - min array(s)

mdptoolbox.util.**isNonNegative**(*matrix*)
Check that `matrix` is row non-negative.

Returns **is_stochastic** – `True` if `matrix` is non-negative, `False` otherwise.

Return type bool

mdptoolbox.util.**isSquare**(*matrix*)
Check that `matrix` is square.

Returns **is_square** – `True` if `matrix` is square, `False` otherwise.

Return type bool

mdptoolbox.util.**isStochastic**(*matrix*)
Check that `matrix` is row stochastic.

Returns **is_stochastic** – `True` if `matrix` is row stochastic, `False` otherwise.

Return type bool

8.4 Markov Decision Process (MDP) Toolbox: `example` module

The `example` module provides functions to generate valid MDP transition and reward matrices.

8.4.1 Available functions

forest() A simple forest management example

rand() A random example

small() A very small example

mdptoolbox.example.**forest**(*S=3*, *r1=4*, *r2=2*, *p=0.1*, *is_sparse=False*)
Generate a MDP example based on a simple forest management scenario.

This function is used to generate a transition probability (`A` × `S` × `S`) array `P` and a reward (`S` × `A`) matrix `R` that model the following problem. A forest is managed by two actions: 'Wait' and 'Cut'. An action is decided each year with first the objective to maintain an old forest for wildlife and second to make money selling cut wood. Each year there is a probability `p` that a fire burns the forest.

Here is how the problem is modelled. Let {0, 1 . . . `S`-1 } be the states of the forest, with `S`-1 being the oldest. Let 'Wait' be action 0 and 'Cut' be action 1. After a fire, the forest is in the youngest state, that is state 0. The transition matrix `P` of the problem can then be defined as follows:

```
              | p 1-p 0.......0  |
              | .  0 1-p 0....0  |
P[0,:,:] =    | .  .  0  .       |
              | .  .        .    |
              | .  .         1-p |
              | p  0  0....0 1-p |
```

```
             | 1 0..........0 |
             | . .            . |
P[1,:,:] =   | . .            . |
             | . .            . |
             | . .            . |
             | 1 0..........0 |
```

The reward matrix R is defined as follows:

```
          |  0  |
          |  .  |
R[:,0] =  |  .  |
          |  .  |
          |  0  |
          |  r1 |

          |  0  |
          |  1  |
R[:,1] =  |  .  |
          |  .  |
          |  1  |
          |  r2 |
```

S [int, optional] The number of states, which should be an integer greater than 1. Default: 3.

r1 [float, optional] The reward when the forest is in its oldest state and action 'Wait' is performed. Default: 4.

r2 [float, optional] The reward when the forest is in its oldest state and action 'Cut' is performed. Default: 2.

p [float, optional] The probability of wild fire occurence, in the range]0, 1[. Default: 0.1.

is_sparse [bool, optional] If True, then the probability transition matrices will be returned in sparse format, otherwise they will be in dense format. Default: False.

Returns **out** – `out[0]` contains the transition probability matrix P and `out[1]` contains the reward matrix R. If `is_sparse=False` then P is a numpy array with a shape of `(A, S, S)` and R is a numpy array with a shape of `(S, A)`. If `is_sparse=True` then P is a tuple of length `A` where each `P[a]` is a scipy sparse CSR format matrix of shape `(S, S)`; R remains the same as in the case of `is_sparse=False`.

Return type tuple

Examples

```
>>> import mdptoolbox.example
>>> P, R = mdptoolbox.example.forest()
>>> P
array([[[ 0.1,  0.9,  0. ],
        [ 0.1,  0. ,  0.9],
        [ 0.1,  0. ,  0.9]],

       [[ 1. ,  0. ,  0. ],
        [ 1. ,  0. ,  0. ],
        [ 1. ,  0. ,  0. ]]])
>>> R
array([[ 0.,  0.],
       [ 0.,  1.],
```

```
       [ 4.,  2.]])
>>> Psp, Rsp = mdptoolbox.example.forest(is_sparse=True)
>>> len(Psp)
2
>>> Psp[0]
<3x3 sparse matrix of type '<... 'numpy.float64'>'
    with 6 stored elements in Compressed Sparse Row format>
>>> Psp[1]
<3x3 sparse matrix of type '<... 'numpy.int64'>'
    with 3 stored elements in Compressed Sparse Row format>
>>> Rsp
array([[ 0.,  0.],
       [ 0.,  1.],
       [ 4.,  2.]])
>>> (Psp[0].todense() == P[0]).all()
True
>>> (Rsp == R).all()
True
```

mdptoolbox.example.**rand**(*S*, *A*, *is_sparse=False*, *mask=None*)

Generate a random Markov Decision Process.

Parameters

- **S** (*int*) – Number of states (> 1)
- **A** (*int*) – Number of actions (> 1)
- **is_sparse** (*bool, optional*) – False to have matrices in dense format, True to have sparse matrices. Default: False.
- **mask** (*array, optional*) – Array with 0 and 1 (0 indicates a place for a zero probability), shape can be `(S, S)` or `(A, S, S)`. Default: random.

Returns **out** – `out[0]` contains the transition probability matrix P and `out[1]` contains the reward matrix R. If `is_sparse=False` then P is a numpy array with a shape of `(A, S, S)` and R is a numpy array with a shape of `(S, A)`. If `is_sparse=True` then P and R are tuples of length `A`, where each `P[a]` is a scipy sparse CSR format matrix of shape `(S, S)` and each `R[a]` is a scipy sparse csr format matrix of shape `(S, 1)`.

Return type tuple

Examples

```
>>> import numpy, mdptoolbox.example
>>> numpy.random.seed(0) # Needed to get the output below
>>> P, R = mdptoolbox.example.rand(4, 3)
>>> P
array([[[ 0.21977283,  0.14889403,  0.30343592,  0.32789723],
        [ 1.        ,  0.        ,  0.        ,  0.        ],
        [ 0.        ,  0.43718772,  0.54480359,  0.01800869],
        [ 0.39766289,  0.39997167,  0.12547318,  0.07689227]],

       [[ 1.        ,  0.        ,  0.        ,  0.        ],
        [ 0.32261337,  0.15483812,  0.32271303,  0.19983549],
        [ 0.33816885,  0.2766999 ,  0.12960299,  0.25552826],
        [ 0.41299411,  0.        ,  0.58369957,  0.00330633]],

       [[ 0.32343037,  0.15178596,  0.28733094,  0.23745272],
```

```
        [ 0.36348538,  0.24483321,  0.16114188,  0.23053953],
        [ 1.        ,  0.        ,  0.        ,  0.        ],
        [ 0.        ,  0.        ,  1.        ,  0.        ]]])
>>> R
array([[[-0.23311696,  0.58345008,  0.05778984,  0.13608912],
        [-0.07704128,  0.        , -0.        ,  0.        ],
        [ 0.        ,  0.22419145,  0.23386799,  0.88749616],
        [-0.3691433 , -0.27257846,  0.14039354, -0.12279697]],

       [[-0.77924972,  0.        , -0.        , -0.        ],
        [ 0.47852716, -0.92162442, -0.43438607, -0.75960688],
        [-0.81211898,  0.15189299,  0.8585924 , -0.3628621 ],
        [ 0.35563307, -0.        ,  0.47038804,  0.92437709]],

       [[-0.4051261 ,  0.62759564, -0.20698852,  0.76220639],
        [-0.9616136 , -0.39685037,  0.32034707, -0.41984479],
        [-0.13716313,  0.        , -0.        , -0.        ],
        [ 0.        , -0.        ,  0.55810204,  0.        ]]])
>>> numpy.random.seed(0) # Needed to get the output below
>>> Psp, Rsp = mdptoolbox.example.rand(100, 5, is_sparse=True)
>>> len(Psp), len(Rsp)
(5, 5)
>>> Psp[0]
<100x100 sparse matrix of type '<... 'numpy.float64'>'
    with 3296 stored elements in Compressed Sparse Row format>
>>> Rsp[0]
<100x100 sparse matrix of type '<... 'numpy.float64'>'
    with 3296 stored elements in Compressed Sparse Row format>
>>> # The number of non-zero elements (nnz) in P and R are equal
>>> Psp[1].nnz == Rsp[1].nnz
True
```

mdptoolbox.example.**small**()

A very small Markov decision process.

The probability transition matrices are:

```
    | | 0.5 0.5 | |
    | | 0.8 0.2 | |
P = |             |
    | | 0.0 1.0 | |
    | | 0.1 0.9 | |
```

The reward matrix is:

```
R = |  5 10 |
    | -1  2 |
```

Returns **out** – `out[0]` is a numpy array of the probability transition matriices. `out[1]` is a numpy arrray of the reward matrix.

Return type tuple

Examples

```
>>> import mdptoolbox.example
>>> P, R = mdptoolbox.example.small()
```

```
>>> P
array([[[ 0.5,  0.5],
        [ 0.8,  0.2]],

       [[ 0. ,  1. ],
        [ 0.1,  0.9]]])
>>> R
array([[ 5, 10],
       [-1,  2]])
```

CHAPTER 9

Indices and tables

- *genindex*
- *modindex*
- *search*

m

C

D

F

G

I

M

P

Q

R

S

T

V