QuantEcon Documentation

Release 0.3.5.1

QuantEcon Developer Team

May 31, 2017

Contents

The *quantecon* python library consists of a number of modules which includes game theory (game_theory), markov chains (markov), random generation utilities (random), a collection of tools (tools), and other utilities (util) which are mainly used by developers internal to the package.

CHAPTER 1

Game theory

1.1 lemke_howson

Author: Daisuke Oyama

Compute mixed Nash equilibria of a 2-player normal form game by the Lemke-Howson algorithm.

`quantecon.game_theory.lemke_howson.`**`lemke_howson`**(*g*, *init_pivot=0*, *max_iter=1000000*, *capping=None*, *full_output=False*)

Find one mixed-action Nash equilibrium of a 2-player normal form game by the Lemke-Howson algorithm *[R2]*, implemented with "complementary pivoting" (see, e.g., von Stengel *[R3]* for details).

Parameters **g** : NormalFormGame

NormalFormGame instance with 2 players.

init_pivot : scalar(int), optional(default=0)

Initial pivot, an integer k such that 0 <= k < m+n, where integers 0, ..., m-1 and m, ..., m+n-1 correspond to the actions of players 0 and 1, respectively.

max_iter : scalar(int), optional(default=10**6)

Maximum number of pivoting steps.

capping : scalar(int), optional(default=None)

If supplied, the routine is executed with the heuristics proposed by Codenotti et al. *[R1]*; see Notes below for details.

full_output : bool, optional(default=False)

If False, only the computed Nash equilibrium is returned. If True, the return value is *(NE, res)*, where *NE* is the Nash equilibrium and *res* is a *NashResult* object.

Returns **NE** : tuple(ndarray(float, ndim=1))

Tuple of computed Nash equilibrium mixed actions.

res : NashResult

Object containing information about the computation. Returned only when *full_output* is True. See *NashResult* for details.

Notes

- This routine is implemented with floating point arithmetic and thus is subject to numerical instability.
- If *capping* is set to a positive integer, the routine is executed with the heuristics proposed by *[R1]*:
  - For k = *init_pivot*, *init_pivot* + 1, ..., *init_pivot* + (m+n-2), (modulo m+n), the Lemke-Howson algorithm is executed with k as the initial pivot and *capping* as the maximum number of pivoting steps. If the algorithm converges during this loop, then the Nash equilibrium found is returned.
  - Otherwise, the Lemke-Howson algorithm is executed with *init_pivot* + (m+n-1) (modulo m+n) as the initial pivot, with a limit *max_iter* on the total number of pivoting steps.

  Accoding to the simulation results for *uniformly random games*, for medium- to large-size games this heuristics outperforms the basic Lemke-Howson algorithm with a fixed initial pivot, where *[R1]* suggests that *capping* be set to 10.

References

[R1], *[R2]*, *[R3]*

Examples

Consider the following game from von Stengel *[R3]*:

```
>>> np.set_printoptions(precision=4)  # Reduce the digits printed
>>> bimatrix = [[(3, 3), (3, 2)],
...             [(2, 2), (5, 6)],
...             [(0, 3), (6, 1)]]
>>> g = NormalFormGame(bimatrix)
```

Obtain a Nash equilibrium of this game by *lemke_howson* with player 0's action 1 (out of the three actions 0, 1, and 2) as the initial pivot:

```
>>> lemke_howson(g, init_pivot=1)
(array([ 0.    ,  0.3333,  0.6667]), array([ 0.3333,  0.6667]))
>>> g.is_nash(_)
True
```

Additional information is returned if *full_output* is set True:

```
>>> NE, res = lemke_howson(g, init_pivot=1, full_output=True)
>>> res.converged  # Whether the routine has converged
True
>>> res.num_iter  # Number of pivoting steps performed
4
```

1.2 mclennan_tourky

Author: Daisuke Oyama

Compute mixed Nash equilibria of an N-player normal form game by applying the imitation game algorithm by McLennan and Tourky to the best response correspondence.

`quantecon.game_theory.mclennan_tourky.`**`mclennan_tourky`**(*g*, *init=None*, *epsilon=0.001*, *max_iter=200*, *full_output=False*)

Find one mixed-action epsilon-Nash equilibrium of an N-player normal form game by the fixed point computation algorithm by McLennan and Tourky *[R4]*.

Parameters **g** : NormalFormGame

NormalFormGame instance.

init : array_like(int or array_like(float, ndim=1)), optional

Initial action profile, an array of N objects, where each object must be an iteger (pure action) or an array of floats (mixed action). If None, default to an array of zeros (the zero-th action for each player).

epsilon : scalar(float), optional(default=1e-3)

Value of epsilon-optimality.

max_iter : scalar(int), optional(default=100)

Maximum number of iterations.

full_output : bool, optional(default=False)

If False, only the computed Nash equilibrium is returned. If True, the return value is *(NE, res)*, where *NE* is the Nash equilibrium and *res* is a *NashResult* object.

Returns **NE** : tuple(ndarray(float, ndim=1))

Tuple of computed Nash equilibrium mixed actions.

res : NashResult

Object containing information about the computation. Returned only when *full_output* is True. See *NashResult* for details.

References

[R4]

Examples

Consider the following version of 3-player "anti-coordination" game, where action 0 is a safe action which yields payoff 1, while action 1 yields payoff v if no other player plays 1 and payoff 0 otherwise:

```
>>> N = 3
>>> v = 2
>>> payoff_array = np.empty((2,)*n)
>>> payoff_array[0, :] = 1
>>> payoff_array[1, :] = 0
>>> payoff_array[1].flat[0] = v
>>> g = NormalFormGame((Player(payoff_array),)*N)
>>> print(g)
3-player NormalFormGame with payoff profile array:
[[[[ 1.,  1.,  1.],   [ 1.,  1.,  2.]],
```

```
 [[ 1.,  2.,  1.],   [ 1.,  0.,  0.]]],
[[[ 2.,  1.,  1.],   [ 0.,  1.,  0.]],
 [[ 0.,  0.,  1.],   [ 0.,  0.,  0.]]]]
```

This game has a unique symmetric Nash equilibrium, where the equilibrium action is given by $(p^*, 1 - p^*)$ with $p^* = 1/v^{1/(N-1)}$:

```
>>> p_star = 1/(v**(1/(N-1)))
>>> [p_star, 1 - p_star]
[0.7071067811865475, 0.29289321881345254]
```

Obtain an approximate Nash equilibrium of this game by *mclennan_tourky*:

```
>>> epsilon = 1e-5  # Value of epsilon-optimality
>>> NE = mclennan_tourky(g, epsilon=epsilon)
>>> print(NE[0], NE[1], NE[2], sep='\n')
[ 0.70710754  0.29289246]
[ 0.70710754  0.29289246]
[ 0.70710754  0.29289246]
>>> g.is_nash(NE, tol=epsilon)
True
```

Additional information is returned if *full_output* is set True:

```
>>> NE, res = mclennan_tourky(g, epsilon=epsilon, full_output=True)
>>> res.converged
True
>>> res.num_iter
18
```

1.3 normal_form_game

Authors: Tomohiro Kusano, Daisuke Oyama

Tools for normal form games.

1.3.1 Definitions and Basic Concepts

An N-player *normal form game* $g = (I, (A_i)_{i \in I}, (u_i)_{i \in I})$ consists of

- the set of *players* $I = \{0, \dots, N - 1\}$,
- the set of *actions* $A_i = \{0, \dots, n_i - 1\}$ for each player $i \in I$, and
- the *payoff function* $u_i \colon A_i \times A_{i+1} \times \cdots \times A_{i+N-1} \to \mathbb{R}$ for each player $i \in I$,

where $i + j$ is understood modulo N. Note that we adopt the convention that the 0-th argument of the payoff function u_i is player i's own action and the j-th argument is player $(i + j)$'s action (modulo N). A mixed action for player i is a probability distribution on A_i (while an element of A_i is referred to as a pure action). A pure action $a_i \in A_i$ is identified with the mixed action that assigns probability one to a_i. Denote the set of mixed actions of player i by X_i. We also denote $A_{-i} = A_{i+1} \times \cdots \times A_{i+N-1}$ and $X_{-i} = X_{i+1} \times \cdots \times X_{i+N-1}$.

The (pure-action) *best response correspondence* $b_i \colon X_{-i} \to A_i$ for each player i is defined by

$$b_i(x_{-i}) = \{a_i \in A_i \mid u_i(a_i, x_{-i}) \geq u_i(a_i', x_{-i}) \,\forall\, a_i' \in A_i\},$$

where $u_i(a_i, x_{-i}) = \sum_{a_{-i} \in A_{-i}} u_i(a_i, a_{-i}) \prod_{j=1}^{N-1} x_{i+j}(a_j)$ is the expected payoff to action a_i against mixed actions x_{-i}. A profile of mixed actions $x^* \in X_0 \times \cdots \times X_{N-1}$ is a *Nash equilibrium* if for all $i \in I$ and $a_i \in A_i$,

$$x_i^*(a_i) > 0 \Rightarrow a_i \in b_i(x_{-i}^*),$$

or equivalently, $x_i^* \cdot v_i(x_{-i}^*) \geq x_i \cdot v_i(x_{-i}^*)$ for all $x_i \in X_i$, where $v_i(x_{-i})$ is the vector of player i's payoffs when the opponent players play mixed actions x_{-i}.

1.3.2 Creating a NormalFormGame

There are three ways to construct a *NormalFormGame* instance.

The first is to pass an array of payoffs for all the players:

```
>>> matching_pennies_bimatrix = [[(1, -1), (-1, 1)], [(-1, 1), (1, -1)]]
>>> g = NormalFormGame(matching_pennies_bimatrix)
>>> print(g.players[0])
Player in a 2-player normal form game with payoff array:
[[ 1, -1],
 [-1,  1]]
>>> print(g.players[1])
Player in a 2-player normal form game with payoff array:
[[-1,  1],
 [ 1, -1]]
```

If a square matrix (2-dimensional array) is given, then it is considered to be a symmetric two-player game:

```
>>> coordination_game_matrix = [[4, 0], [3, 2]]
>>> g = NormalFormGame(coordination_game_matrix)
>>> print(g)
2-player NormalFormGame with payoff profile array:
[[[4, 4],  [0, 3]],
 [[3, 0],  [2, 2]]]
```

The second is to specify the sizes of the action sets of the players, which gives a *NormalFormGame* instance filled with payoff zeros, and then set the payoff values to each entry:

```
>>> g = NormalFormGame((2, 2))
>>> print(g)
2-player NormalFormGame with payoff profile array:
[[[ 0.,   0.],   [ 0.,   0.]],
 [[ 0.,   0.],   [ 0.,   0.]]]
>>> g[0, 0] = 1, 1
>>> g[0, 1] = -2, 3
>>> g[1, 0] = 3, -2
>>> print(g)
2-player NormalFormGame with payoff profile array:
[[[ 1.,   1.],   [-2.,   3.]],
 [[ 3.,  -2.],   [ 0.,   0.]]]
```

The third is to pass an array of *Player* instances, as explained in the next section.

1.3.3 Creating a Player

A *Player* instance is created by passing a payoff array:

```
>>> player0 = Player([[3, 1], [0, 2]])
>>> player0.payoff_array
array([[3, 1],
       [0, 2]])
```

Passing an array of *Player* instances is the third way to create a *NormalFormGame* instance.

```
>>> player1 = Player([[2, 0], [1, 3]])
>>> player1.payoff_array
array([[2, 0],
       [1, 3]])
>>> g = NormalFormGame((player0, player1))
>>> print(g)
2-player NormalFormGame with payoff profile array:
[[[3, 2],  [1, 1]],
 [[0, 0],  [2, 3]]]
```

Beware that in *payoff_array[h, k]*, *h* refers to the player's own action, while *k* refers to the opponent player's action.

class `quantecon.game_theory.normal_form_game.`**`NormalFormGame`**(*data*, *dtype=None*)
Bases: `object`

Class representing an N-player normal form game.

Parameters **data** : array_like of Player, int (ndim=1), or float (ndim=2 or N+1)

Data to initialize a NormalFormGame. *data* may be an array of Players, in which case the shapes of the Players' payoff arrays must be consistent. If *data* is an array of N integers, then these integers are treated as the numbers of actions of the N players and a NormalFormGame is created consisting of payoffs all 0 with *data[i]* actions for each player *i*. *data* may also be an (N+1)-dimensional array representing payoff profiles. If *data* is a square matrix (2-dimensional array), then the game will be a symmetric two-player game where the payoff matrix of each player is given by the input matrix.

dtype : data-type, optional(default=None)

Relevant only when *data* is an array of integers. Data type of the players' payoff arrays. If not supplied, default to numpy.float64.

Attributes

players	(tuple(Player)) Tuple of the Player instances of the game.
N	(scalar(int)) The number of players.
nums_actions	(tuple(int)) Tuple of the numbers of actions, one for each player.

Methods

is_nash(action_profile[, tol])	Return True if *action_profile* is a Nash equilibrium.

`is_nash`(*action_profile*, *tol=None*)
Return True if *action_profile* is a Nash equilibrium.

Parameters **action_profile** : array_like(int or array_like(float))

An array of N objects, where each object must be an integer (pure action) or an array of floats (mixed action).

tol : scalar(float)

Tolerance level used in determining best responses. If None, default to each player's *tol* attribute value.

Returns bool

True if *action_profile* is a Nash equilibrium; False otherwise.

payoff_profile_array

class `quantecon.game_theory.normal_form_game.Player`(*payoff_array*)
Bases: `object`

Class representing a player in an N-player normal form game.

Parameters **payoff_array** : array_like(float)

Array representing the player's payoff function, where *payoff_array[a_0, a_1, ..., a_{N-1}]* is the payoff to the player when the player plays action *a_0* while his N-1 opponents play actions *a_1*, ..., *a_{N-1}*, respectively.

Attributes

payoff_array	(ndarray(float, ndim=N)) See Parameters.
num_actions	(scalar(int)) The number of actions available to the player.
num_opponents	(scalar(int)) The number of opponent players.
dtype	(dtype) Data type of the elements of *payoff_array*.
tol	(scalar(float), default=1e-8) Default tolerance value used in determining best responses.

Methods

`best_response`(opponents_actions[, ...])	Return the best response action(s) to *opponents_actions*.
`is_best_response`(own_action, opponents_actions)	Return True if *own_action* is a best response to *opponents_actions*.
`payoff_vector`(opponents_actions)	Return an array of payoff values, one for each own action, given a profile of the opponents' actions.
`random_choice`([actions, random_state])	Return a pure action chosen randomly from *actions*.

best_response(*opponents_actions*, *tie_breaking='smallest'*, *payoff_perturbation=None*, *tol=None*, *random_state=None*)
Return the best response action(s) to *opponents_actions*.

Parameters **opponents_actions** : scalar(int) or array_like

A profile of N-1 opponents' actions, represented by either scalar(int), array_like(float), array_like(int), or array_like(array_like(float)). If N=2, then it must be a scalar of integer (in which case it is treated as the opponent's pure action) or a 1-dimensional array of floats (in which case it is treated as the opponent's mixed action). If N>2, then it must be an array of N-1 objects, where each object must be an integer (pure action) or an array of floats (mixed action).

tie_breaking : str, optional(default='smallest')

str in {'smallest', 'random', False}. Control how, or whether, to break a tie (see Returns for details).

payoff_perturbation : array_like(float), optional(default=None)

Array of length equal to the number of actions of the player containing the values ("noises") to be added to the payoffs in determining the best response.

tol : scalar(float), optional(default=None)

Tolerance level used in determining best responses. If None, default to the value of the *tol* attribute.

random_state : int or np.random.RandomState, optional

Random seed (integer) or np.random.RandomState instance to set the initial state of the random number generator for reproducibility. If None, a randomly initialized RandomState is used. Relevant only when tie_breaking='random'.

Returns scalar(int) or ndarray(int, ndim=1)

If tie_breaking=False, returns an array containing all the best response pure actions. If tie_breaking='smallest', returns the best response action with the smallest index; if tie_breaking='random', returns an action randomly chosen from the best response actions.

is_best_response(*own_action*, *opponents_actions*, *tol=None*)

Return True if *own_action* is a best response to *opponents_actions*.

Parameters **own_action** : scalar(int) or array_like(float, ndim=1)

An integer representing a pure action, or an array of floats representing a mixed action.

opponents_actions : see *best_response*

tol : scalar(float), optional(default=None)

Tolerance level used in determining best responses. If None, default to the value of the *tol* attribute.

Returns bool

True if *own_action* is a best response to *opponents_actions*; False otherwise.

payoff_vector(*opponents_actions*)

Return an array of payoff values, one for each own action, given a profile of the opponents' actions.

Parameters **opponents_actions** : see *best_response*.

Returns **payoff_vector** : ndarray(float, ndim=1)

An array representing the player's payoff vector given the profile of the opponents' actions.

random_choice(*actions=None*, *random_state=None*)

Return a pure action chosen randomly from *actions*.

Parameters **actions** : array_like(int), optional(default=None)

An array of integers representing pure actions.

random_state : int or np.random.RandomState, optional

Random seed (integer) or np.random.RandomState instance to set the initial state of the random number generator for reproducibility. If None, a randomly initialized RandomState is used.

Returns scalar(int)

If *actions* is given, returns an integer representing a pure action chosen randomly from *actions*; if not, an action is chosen randomly from the player's all actions.

`quantecon.game_theory.normal_form_game.`**`best_response_2p`**

Numba-optimized version of *Player.best_response* compiled in nopython mode, specialized for 2-player games (where there is only one opponent).

Return the best response action (with the smallest index if more than one) to *opponent_mixed_action* under *payoff_matrix*.

Parameters **payoff_matrix** : ndarray(float, ndim=2)

Payoff matrix.

opponent_mixed_action : ndarray(float, ndim=1)

Opponent's mixed action. Its length must be equal to *payoff_matrix.shape[1]*.

tol : scalar(float), optional(default=None)

Tolerance level used in determining best responses.

Returns scalar(int)

Best response action.

`quantecon.game_theory.normal_form_game.`**`pure2mixed`**(*num_actions*, *action*)

Convert a pure action to the corresponding mixed action.

Parameters **num_actions** : scalar(int)

The number of the pure actions (= the length of a mixed action).

action : scalar(int)

The pure action to convert to the corresponding mixed action.

Returns ndarray(float, ndim=1)

The mixed action representation of the given pure action.

1.4 pure_nash

Author: Zejin Shi

Methods for computing pure Nash equilibria of a normal form game. (For now, only brute force method is supported)

`quantecon.game_theory.pure_nash.`**`pure_nash_brute`**(*g*)

Find all pure Nash equilibria of a normal form game by brute force.

Parameters **g** : NormalFormGame

Returns **NEs** : list(tuple(int))

List of tuples of Nash equilibrium pure actions. If no pure Nash equilibrium is found, return empty list.

Examples

Consider the "Prisoners' Dilemma" game:

```
>>> PD_bimatrix = [[(1, 1), (-2, 3)],
...                [(3, -2), (0, 0)]]
>>> g_PD = NormalFormGame(PD_bimatrix)
>>> pure_nash_brute(g_PD)
[(1, 1)]
```

If we consider the "Matching Pennies" game, which has no pure nash equilibirum:

```
>>> MP_bimatrix = [[(1, -1), (-1, 1)],
...                [(-1, 1), (1, -1)]]
>>> g_MP = NormalFormGame(MP_bimatrix)
>>> pure_nash_brute(g_MP)
[]
```

quantecon.game_theory.pure_nash.**pure_nash_brute_gen**(*g*)

Generator version of *pure_nash_brute*.

Parameters g : NormalFormGame

Yields out : tuple(int)

Tuple of Nash equilibrium pure actions.

1.5 random

Filename: random.py

Author: Daisuke Oyama

Generate random NormalFormGame instances.

quantecon.game_theory.random.**covariance_game**(*nums_actions*, *rho*, *random_state=None*)

Return a random NormalFormGame instance where the payoff profiles are drawn independently from the standard multi-normal with the covariance of any pair of payoffs equal to *rho*, as studied in *[R5]*.

Parameters nums_actions : tuple(int)

Tuple of the numbers of actions, one for each player.

rho : scalar(float)

Covariance of a pair of payoff values. Must be in [-1/(N-1), 1], where N is the number of players.

random_state : int or np.random.RandomState, optional

Random seed (integer) or np.random.RandomState instance to set the initial state of the random number generator for reproducibility. If None, a randomly initialized RandomState is used.

Returns g : NormalFormGame

References

[R5]

quantecon.game_theory.random.**random_game**(*nums_actions*, *random_state=None*)

Return a random NormalFormGame instance where the payoffs are drawn independently from the uniform distribution on [0, 1).

Parameters **nums_actions** : tuple(int)

Tuple of the numbers of actions, one for each player.

random_state : int or np.random.RandomState, optional

Random seed (integer) or np.random.RandomState instance to set the initial state of the random number generator for reproducibility. If None, a randomly initialized RandomState is used.

Returns **g** : NormalFormGame

1.6 support_enumeration

Author: Daisuke Oyama

Compute all mixed Nash equilibria of a 2-player (non-degenerate) normal form game by support enumeration.

1.6.1 References

B. von Stengel, "Equilibrium Computation for Two-Player Games in Strategic and Extensive Form," Chapter 3, N. Nisan, T. Roughgarden, E. Tardos, and V. Vazirani eds., Algorithmic Game Theory, 2007.

`quantecon.game_theory.support_enumeration.`**`support_enumeration`**(*g*)

Compute mixed-action Nash equilibria with equal support size for a 2-player normal form game by support enumeration. For a non-degenerate game input, these are all the Nash equilibria.

The algorithm checks all the equal-size support pairs; if the players have the same number n of actions, there are 2n choose n minus 1 such pairs. This should thus be used only for small games.

Parameters **g** : NormalFormGame

NormalFormGame instance with 2 players.

Returns list(tuple(ndarray(float, ndim=1)))

List containing tuples of Nash equilibrium mixed actions.

`quantecon.game_theory.support_enumeration.`**`support_enumeration_gen`**(*g*)

Generator version of *support_enumeration*.

Parameters **g** : NormalFormGame

NormalFormGame instance with 2 players.

Yields tuple(ndarray(float, ndim=1))

Tuple of Nash equilibrium mixed actions.

1.7 utilities

Utility routines for the game_theory submodule

class `quantecon.game_theory.utilities.`**`NashResult`**

Bases: `dict`

Contain the information about the result of Nash equilibrium computation.

Notes

This is sourced from *sicpy.optimize.OptimizeResult*.

There may be additional attributes not listed above depending of the routine.

Attributes

NE	(tuple(ndarray(float, ndim=1))) Computed Nash equilibrium.
converged	(bool) Whether the routine has converged.
num_iter	(int) Number of iterations.
max_iter	(int) Maximum number of iterations.
init	(scalar or array_like) Initial condition used.

Methods

`clear`(() -> None. Remove all items from D.)	
`copy`(() -> a shallow copy of D)	
`fromkeys`	Returns a new dict with keys from iterable and values equal to value.
`get`((k[,d]) -> D[k] if k in D, ...)	
`items`(...)	
`keys`(...)	
`pop`((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise KeyError is raised
`popitem`(() -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
`setdefault`((k[,d]) -> D.get(k,d), ...)	
`update`(([E, ...)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
`values`(...)	

CHAPTER 2

Markov

2.1 approximation

Filename: approximation.py

Authors: Thomas Sargent, John Stachurski

2.1.1 tauchen

Discretizes Gaussian linear AR(1) processes via Tauchen's method

`quantecon.markov.approximation.`**`rouwenhorst`**(*n*, *ybar*, *sigma*, *rho*)

Takes as inputs n, p, q, psi. It will then construct a markov chain that estimates an AR(1) process of: y_t = ar{y} +

ho y_{t-1} + arepsilon_t

where

arepsilon_t is i.i.d. normal of mean 0, std dev of sigma

The Rouwenhorst approximation uses the following recursive defintion for approximating a distribution:

theta_2 = [p , 1 - p] [1 - q, q]

theta_{n+1} = p [theta_n, 0] + (1 - p) [0, theta_n]

[0 , 0] [0, 0]

- **q [0 , 0] + (1 - q) [0,]** [theta_n , 0] [0, theta_n]

Parameters **n** : int

The number of points to approximate the distribution

ybar [float] The value ar{y} in the process. Note that the mean of this AR(1) process, y, is simply ybar/(1 - rho)

sigma [float] The value of the standard deviation of the

arepsilon process

rho [float] By default this will be 0, but if you are approximating an AR(1) process then this is the autocorrelation across periods

Returns mc : MarkovChain

An instance of the MarkovChain class that stores the transition matrix and state values returned by the discretization method

`quantecon.markov.approximation.`**`std_norm_cdf`**

`quantecon.markov.approximation.`**`tauchen`**(*rho*, *sigma_u*, *m=3*, *n=7*)

Computes a Markov chain associated with a discretized version of the linear Gaussian AR(1) process

y_{t+1} = rho * y_t + u_{t+1}

using Tauchen's method. Here {u_t} is an iid Gaussian process with zero mean.

Parameters rho : scalar(float)

The autocorrelation coefficient

sigma_u : scalar(float)

The standard deviation of the random process

m : scalar(int), optional(default=3)

The number of standard deviations to approximate out to

n : scalar(int), optional(default=7)

The number of states to use in the approximation

Returns mc : MarkovChain

An instance of the MarkovChain class that stores the transition matrix and state values returned by the discretization method

2.2 core

Authors: Chase Coleman, Spencer Lyon, Daisuke Oyama, Tom Sargent, John Stachurski

Filename: core.py

This file contains some useful objects for handling a finite-state discrete-time Markov chain.

2.2.1 Definitions and Some Basic Facts about Markov Chains

Let $\{X_t\}$ be a Markov chain represented by an $n \times n$ stochastic matrix P. State i *has access* to state j, denoted $i \to j$, if $i = j$ or $P^k[i, j] > 0$ for some $k = 1, 2, \ldots$; i and j *communicate*, denoted $i \leftrightarrow j$, if $i \to j$ and $j \to i$. The binary relation $\leftrightarrow$ is an equivalent relation. A *communication class* of the Markov chain $\{X_t\}$, or of the stochastic matrix P, is an equivalent class of $\leftrightarrow$. Equivalently, a communication class is a *strongly connected component* (SCC) in the associated *directed graph* $\Gamma(P)$, a directed graph with n nodes where there is an edge from i to j if and only if

$P[i, j] > 0$. The Markov chain, or the stochastic matrix, is *irreducible* if it admits only one communication class, or equivalently, if $\Gamma(P)$ is *strongly connected*.

A state i is *recurrent* if $i \to j$ implies $j \to i$; it is *transient* if it is not recurrent. For any i, j contained in a communication class, i is recurrent if and only if j is recurrent. Therefore, recurrence is a property of a communication class. Thus, a communication class is a *recurrent class* if it contains a recurrent state. Equivalently, a recurrent class is a SCC that corresponds to a sink node in the *condensation* of the directed graph $\Gamma(P)$, where the condensation of $\Gamma(P)$ is a directed graph in which each SCC is replaced with a single node and there is an edge from one SCC C to another SCC C' if $C \neq C'$ and some node in C has access to some node in C'. A recurrent class is also called a *closed communication class*. The condensation is acyclic, so that there exists at least one recurrent class.

For example, if the entries of P are all strictly positive, then the whole state space is a communication class as well as a recurrent class. (More generally, if there is only one communication class, then it is a recurrent class.) As another example, consider the stochastic matrix $P = [[1, 0], [0, 5, 0.5]]$. This has two communication classes, $\{0\}$ and $\{1\}$, and $\{0\}$ is the only recurrent class.

A *stationary distribution* of the Markov chain $\{X_t\}$, or of the stochastic matrix P, is a nonnegative vector x such that $x'P = x'$ and $x'\mathbf{1} = 1$, where $\mathbf{1}$ is the vector of ones. The Markov chain has a unique stationary distribution if and only if it has a unique recurrent class. More generally, each recurrent class has a unique stationary distribution whose support equals that recurrent class. The set of all stationary distributions is given by the convex hull of these unique stationary distributions for the recurrent classes.

A natural number d is the *period* of state i if it is the greatest common divisor of all k's such that $P^k[i, i] > 0$; equivalently, it is the GCD of the lengths of the cycles in $\Gamma(P)$ passing through i. For any i, j contained in a communication class, i has period d if and only if j has period d. The *period* of an irreducible Markov chain (or of an irreducible stochastic matrix) is the period of any state. We define the period of a general (not necessarily irreducible) Markov chain to be the least common multiple of the periods of its recurrent classes, where the period of a recurrent class is the period of any state in that class. A Markov chain is *aperiodic* if its period is one. A Markov chain is irreducible and aperiodic if and only if it is *uniformly ergodic*, i.e., there exists some m such that $P^m[i, j] > 0$ for all i, j (in this case, P is also called *primitive*).

Suppose that an irreducible Markov chain has period d. Fix any state, say state 0. For each $m = 0, \dots, d-1$, let S_m be the set of states i such that $P^{kd+m}[0, i] > 0$ for some k. These sets $S_0, \dots, S_{d-1}$ constitute a partition of the state space and are called the *cyclic classes*. For each S_m and each $i \in S_m$, we have $\sum_{j \in S_{m+1}} P[i, j] = 1$, where $S_d = S_0$.

class `quantecon.markov.core.`**`MarkovChain`**(*P*, *state_values=None*)

Bases: `object`

Class for a finite-state discrete-time Markov chain. It stores useful information such as the stationary distributions, and communication, recurrent, and cyclic classes, and allows simulation of state transitions.

Parameters **P** : array_like or scipy sparse matrix (float, ndim=2)

The transition matrix. Must be of shape n x n.

state_values : array_like(default=None)

Array_like of length n containing the values associated with the states, which must be homogeneous in type. If None, the values default to integers 0 through n-1.

Notes

In computing stationary distributions, if the input matrix is a sparse matrix, internally it is converted to a dense matrix.

Attributes

P	(ndarray or scipy.sparse.csr_matrix (float, ndim=2)) See Parameters
stationary_distributions	(array_like(float, ndim=2)) Array containing stationary distributions, one for each recurrent class, as rows.
is_irreducible	(bool) Indicate whether the Markov chain is irreducible.
num_communication_classes	(int) The number of the communication classes.
communication_classes_indices	(list(ndarray(int))) List of numpy arrays containing the indices of the communication classes.
communication_classes	(list(ndarray)) List of numpy arrays containing the communication classes, where the states are annotated with their values (if *state_values* is not None).
num_recurrent_classes	(int) The number of the recurrent classes.
recurrent_classes_indices	(list(ndarray(int))) List of numpy arrays containing the indices of the recurrent classes.
recurrent_classes	(list(ndarray)) List of numpy arrays containing the recurrent classes, where the states are annotated with their values (if *state_values* is not None).
is_aperiodic	(bool) Indicate whether the Markov chain is aperiodic.
period	(int) The period of the Markov chain.
cyclic_classes_indices	(list(ndarray(int))) List of numpy arrays containing the indices of the cyclic classes. Defined only when the Markov chain is irreducible.
cyclic_classes	(list(ndarray)) List of numpy arrays containing the cyclic classes, where the states are annotated with their values (if *state_values* is not None). Defined only when the Markov chain is irreducible.

Methods

`get_index`(value)	Return the index (or indices) of the given value (or values) in *state_values*.
`simulate`(ts_length[, init, num_reps, ...])	Simulate time series of state transitions, where the states are annotated with their values (if *state_values* is not None).
`simulate_indices`(ts_length[, init, ...])	Simulate time series of state transitions, where state indices are returned.

cdfs

cdfs1d

communication_classes

communication_classes_indices

cyclic_classes

cyclic_classes_indices

digraph

get_index (*value*)

Return the index (or indices) of the given value (or values) in *state_values*.

Parameters value

Value(s) to get the index (indices) for.

Returns idx : int or ndarray(int)

Index of *value* if *value* is a single state value; array of indices if *value* is an array_like of state values.

is_aperiodic

is_irreducible

num_communication_classes

num_recurrent_classes

period

recurrent_classes

recurrent_classes_indices

simulate (*ts_length*, *init=None*, *num_reps=None*, *random_state=None*)

Simulate time series of state transitions, where the states are annotated with their values (if *state_values* is not None).

Parameters **ts_length** : scalar(int)

Length of each simulation.

init : scalar or array_like, optional(default=None)

Initial state values(s). If None, the initial state is randomly drawn.

num_reps : scalar(int), optional(default=None)

Number of repetitions of simulation.

random_state : int or np.random.RandomState, optional

Random seed (integer) or np.random.RandomState instance to set the initial state of the random number generator for reproducibility. If None, a randomly initialized RandomState is used.

Returns **X** : ndarray(ndim=1 or 2)

Array containing the state values of the sample path(s). See the *simulate* method for more information.

simulate_indices (*ts_length*, *init=None*, *num_reps=None*, *random_state=None*)

Simulate time series of state transitions, where state indices are returned.

Parameters **ts_length** : scalar(int)

Length of each simulation.

init : int or array_like(int, ndim=1), optional

Initial state(s). If None, the initial state is randomly drawn.

num_reps : scalar(int), optional(default=None)

Number of repetitions of simulation.

random_state : int or np.random.RandomState, optional

Random seed (integer) or np.random.RandomState instance to set the initial state of the random number generator for reproducibility. If None, a randomly initialized RandomState is used.

Returns **X** : ndarray(ndim=1 or 2)

Array containing the sample path(s), of shape (ts_length,) if init is a scalar (integer) or None and num_reps is None; of shape (k, ts_length) otherwise, where k = len(init) if (init, num_reps) = (array, None), k = num_reps if (init, num_reps) = (int or None, int), and k = len(init)*num_reps if (init, num_reps) = (array, int).

state_values

stationary_distributions

quantecon.markov.core.**mc_compute_stationary**(*P*)

Computes stationary distributions of P, one for each recurrent class. Any stationary distribution is written as a convex combination of these distributions.

Returns **stationary_dists** : array_like(float, ndim=2)

Array containing the stationary distributions as its rows.

quantecon.markov.core.**mc_sample_path**(*P*, *init=0*, *sample_size=1000*, *random_state=None*)

Generates one sample path from the Markov chain represented by (n x n) transition matrix P on state space S = {{0,...,n-1}}.

Parameters **P** : array_like(float, ndim=2)

A Markov transition matrix.

init : array_like(float ndim=1) or scalar(int), optional(default=0)

If init is an array_like, then it is treated as the initial distribution across states. If init is a scalar, then it treated as the deterministic initial state.

sample_size : scalar(int), optional(default=1000)

The length of the sample path.

random_state : int or np.random.RandomState, optional

Random seed (integer) or np.random.RandomState instance to set the initial state of the random number generator for reproducibility. If None, a randomly initialized RandomState is used.

Returns **X** : array_like(int, ndim=1)

The simulation of states.

2.3 ddp

Filename: ddp.py

Author: Daisuke Oyama

Module for solving dynamic programs (also known as Markov decision processes) with finite states and actions.

2.3.1 Discrete Dynamic Programming

A discrete dynamic program consists of the following components:

- finite set of states $S = \{0, \dots, n-1\}$;
- finite set of available actions $A(s)$ for each state $s \in S$ with $A = \bigcup_{s \in S} A(s) = \{0, \dots, m-1\}$, where $SA = \{(s, a) \in S \times A \mid a \in A(s)\}$ is the set of feasible state-action pairs;
- reward function $r \colon SA \to \mathbb{R}$, where $r(s, a)$ is the reward when the current state is s and the action chosen is a;

- transition probability function $q\colon SA \to \Delta(S)$, where $q(s'|s, a)$ is the probability that the state in the next period is s' when the current state is s and the action chosen is a; and
- discount factor $0 \leq \beta < 1$.

For a policy function σ, let r_σ and Q_σ be the reward vector and the transition probability matrix for σ, which are defined by $r_\sigma(s) = r(s, \sigma(s))$ and $Q_\sigma(s, s') = q(s'|s, \sigma(s))$, respectively. The policy value function v_σ for σ is defined by

$$v_\sigma(s) = \sum_{t=0}^{\infty} \beta^t (Q_\sigma^t r_\sigma)(s) \quad (s \in S).$$

The *optimal value function* v^* is the function such that $v^*(s) = \max_\sigma v_\sigma(s)$ for all $s \in S$. A policy function σ^* is *optimal* if $v_{\sigma^*}(s) = v^*(s)$ for all $s \in S$.

The *Bellman equation* is written as

$$v(s) = \max_{a \in A(s)} r(s, a) + \beta \sum_{s' \in S} q(s'|s, a) v(s') \quad (s \in S).$$

The *Bellman operator* T is defined by the right hand side of the Bellman equation:

$$(Tv)(s) = \max_{a \in A(s)} r(s, a) + \beta \sum_{s' \in S} q(s'|s, a) v(s') \quad (s \in S).$$

For a policy function σ, the operator T_σ is defined by

$$(T_\sigma v)(s) = r(s, \sigma(s)) + \beta \sum_{s' \in S} q(s'|s, \sigma(s)) v(s') \quad (s \in S),$$

or $T_\sigma v = r_\sigma + \beta Q_\sigma v$.

The main result of the theory of dynamic programming states that the optimal value function v^* is the unique solution to the Bellman equation, or the unique fixed point of the Bellman operator, and that σ^* is an optimal policy function if and only if it is v^*-greedy, i.e., it satisfies $Tv^* = T_{\sigma^*} v^*$.

2.3.2 Solution Algorithms

The *DiscreteDP* class currently implements the following solution algorithms:

- value iteration;
- policy iteration;
- modified policy iteration.

Policy iteration computes an exact optimal policy in finitely many iterations, while value iteration and modified policy iteration return an ε-optimal policy and an $\varepsilon/2$-approximation of the optimal value function for a prespecified value of ε.

Our implementations of value iteration and modified policy iteration employ the norm-based and span-based termination rules, respectively.

- Value iteration is terminated when the condition $\|Tv - v\| < [(1 - \beta)/(2\beta)]\varepsilon$ is satisfied.
- Modified policy iteration is terminated when the condition $\mathrm{span}(Tv - v) < [(1 - \beta)/\beta]\varepsilon$ is satisfied, where $\mathrm{span}(z) = \max(z) - \min(z)$.

2.3.3 References

M. L. Puterman, Markov Decision Processes: Discrete Stochastic Dynamic Programming, Wiley-Interscience, 2005.

class `quantecon.markov.ddp.`**`DPSolveResult`**

Bases: `dict`

Contain the information about the dynamic programming result.

Attributes

v	(ndarray(float, ndim=1)) Computed optimal value function
sigma	(ndarray(int, ndim=1)) Computed optimal policy function
num_iter	(int) Number of iterations
mc	(MarkovChain) Controlled Markov chain
method	(str) Method employed
epsilon	(float) Value of epsilon
max_iter	(int) Maximum number of iterations

Methods

`clear`(() -> None. Remove all items from D.)	
`copy`(() -> a shallow copy of D)	
`fromkeys`	Returns a new dict with keys from iterable and values equal to value.
`get`((k[,d]) -> D[k] if k in D, ...)	
`items`(...)	
`keys`(...)	
`pop`((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise KeyError is raised
`popitem`(() -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
`setdefault`((k[,d]) -> D.get(k,d), ...)	
`update`(([E, ...)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
`values`(...)	

class `quantecon.markov.ddp.`**`DiscreteDP`**(*R*, *Q*, *beta*, *s_indices=None*, *a_indices=None*)

Bases: `object`

Class for dealing with a discrete dynamic program.

There are two ways to represent the data for instantiating a *DiscreteDP* object. Let n, m, and L denote the numbers of states, actions, and feasbile state-action pairs, respectively.

1. *DiscreteDP(R, Q, beta)*

   with parameters:

   - n x m reward array *R*,
   - n x m x n transition probability array *Q*, and
   - discount factor beta,

where *R[s, a]* is the reward for action *a* when the state is *s* and *Q[s, a, s_next]* is the probability that the state in the next period is *s_next* when the current state is *s* and the action chosen is *a*.

2. *DiscreteDP(R, Q, beta, s_indices, a_indices)*

   with parameters:

   - length L reward vector *R*,
   - L x n transition probability array *Q*,
   - discount factor *beta*,
   - length L array *s_indices*, and
   - length L array *a_indices*,

   where the pairs (*s_indices[0]*, *a_indices[0]*), ..., (*s_indices[L-1]*, *a_indices[L-1]*) enumerate feasible state-action pairs, and *R[i]* is the reward for action *a_indices[i]* when the state is *s_indices[i]* and *Q[i, s_next]* is the probability that the state in the next period is *s_next* when the current state is *s_indices[i]* and the action chosen is *a_indices[i]*. With this formulation, *Q* may be represented by a scipy.sparse matrix.

Parameters **R** : array_like(float, ndim=2 or 1)

Reward array.

Q : array_like(float, ndim=3 or 2) or scipy.sparse matrix

Transition probability array.

beta : scalar(float)

Discount factor. Must be in [0, 1].

s_indices : array_like(int, ndim=1), optional(default=None)

Array containing the indices of the states.

a_indices : array_like(int, ndim=1), optional(default=None)

Array containing the indices of the actions.

Notes

DiscreteDP accepts beta=1 for convenience. In this case, infinite horizon solution methods are disabled, and the instance is then seen as merely an object carrying the Bellman operator, which may be used for backward induction for finite horizon problems.

Examples

Consider the following example, taken from Puterman (2005), Section 3.1, pp.33-35.

- Set of states S = {0, 1}
- Set of actions A = {0, 1}
- Set of feasible state-action pairs SA = {(0, 0), (0, 1), (1, 0)}
- Rewards r(s, a):

  r(0, 0) = 5, r(0, 1) =10, r(1, 0) = -1

- Transition probabilities q(s_next|s, a):

q(0|0, 0) = 0.5, q(1|0, 0) = 0.5, q(0|0, 1) = 0, q(1|0, 1) = 1, q(0|1, 0) = 0, q(1|1, 0) = 1

- Discount factor 0.95

Creating a 'DiscreteDP' instance

Product formulation

This approach uses the product set S x A as the domain by treating action 1 as yielding a reward negative infinity at state 1.

```
>>> R = [[5, 10], [-1, -float('inf')]]
>>> Q = [[(0.5, 0.5), (0, 1)], [(0, 1), (0.5, 0.5)]]
>>> beta = 0.95
>>> ddp = DiscreteDP(R, Q, beta)
```

(*Q[1, 1]* is an arbitrary probability vector.)

State-action pairs formulation

This approach takes the set of feasible state-action pairs SA as given.

```
>>> s_indices = [0, 0, 1]  # State indices
>>> a_indices = [0, 1, 0]  # Action indices
>>> R = [5, 10, -1]
>>> Q = [(0.5, 0.5), (0, 1), (0, 1)]
>>> beta = 0.95
>>> ddp = DiscreteDP(R, Q, beta, s_indices, a_indices)
```

Solving the model

Policy iteration

```
>>> res = ddp.solve(method='policy_iteration', v_init=[0, 0])
>>> res.sigma  # Optimal policy function
array([0, 0])
>>> res.v  # Optimal value function
array([ -8.57142857, -20.        ])
>>> res.num_iter  # Number of iterations
2
```

Value iteration

```
>>> res = ddp.solve(method='value_iteration', v_init=[0, 0],
...                 epsilon=0.01)
>>> res.sigma  # (Approximate) optimal policy function
array([0, 0])
>>> res.v  # (Approximate) optimal value function
array([ -8.5665053 , -19.99507673])
>>> res.num_iter  # Number of iterations
162
```

Modified policy iteration

```
>>> res = ddp.solve(method='modified_policy_iteration',
...                 v_init=[0, 0], epsilon=0.01)
>>> res.sigma  # (Approximate) optimal policy function
array([0, 0])
>>> res.v  # (Approximate) optimal value function
array([ -8.57142826, -19.99999965])
```

```
>>> res.num_iter  # Number of iterations
3
```

Attributes

R, Q, beta	(see Parameters.)
num_states	(scalar(int)) Number of states.
num_sa_pairs	(scalar(int)) Number of feasible state-action pairs (or those that yield finite rewards).
epsilon	(scalar(float), default=1e-3) Default value for epsilon-optimality.
max_iter	(scalar(int), default=250) Default value for the maximum number of iterations.

Methods

`RQ_sigma`(sigma)	Given a policy *sigma*, return the reward vector *R_sigma* and the transition probability matrix *Q_sigma*.
`T_sigma`(sigma)	Given a policy *sigma*, return the T_sigma operator.
`bellman_operator`(v[, Tv, sigma])	The Bellman operator, which computes and returns the updated value function *Tv* for a value function *v*.
`compute_greedy`(v[, sigma])	Compute the v-greedy policy.
`controlled_mc`(sigma)	Returns the controlled Markov chain for a given policy *sigma*.
`evaluate_policy`(sigma)	Compute the value of a policy.
`modified_policy_iteration`([v_init, epsilon, ...])	Solve the optimization problem by modified policy iteration.
`operator_iteration`(T, v, max_iter[, tol])	Iteratively apply the operator *T* to *v*.
`policy_iteration`([v_init, max_iter])	Solve the optimization problem by policy iteration.
`solve`([method, v_init, epsilon, max_iter, k])	Solve the dynamic programming problem.
`value_iteration`([v_init, epsilon, max_iter])	Solve the optimization problem by value iteration.

RQ_sigma (*sigma*)

Given a policy *sigma*, return the reward vector *R_sigma* and the transition probability matrix *Q_sigma*.

Parameters **sigma** : array_like(int, ndim=1)

Policy vector, of length n.

Returns **R_sigma** : ndarray(float, ndim=1)

Reward vector for *sigma*, of length n.

Q_sigma : ndarray(float, ndim=2)

Transition probability matrix for *sigma*, of shape (n, n).

T_sigma (*sigma*)

Given a policy *sigma*, return the T_sigma operator.

Parameters **sigma** : array_like(int, ndim=1)

Policy vector, of length n.

Returns callable

The T_sigma operator.

bellman_operator(*v*, *Tv=None*, *sigma=None*)
The Bellman operator, which computes and returns the updated value function *Tv* for a value function *v*.

Parameters **v** : array_like(float, ndim=1)

Value function vector, of length n.

Tv : ndarray(float, ndim=1), optional(default=None)

Optional output array for Tv.

sigma : ndarray(int, ndim=1), optional(default=None)

If not None, the v-greedy policy vector is stored in this array. Must be of length n.

Returns **Tv** : ndarray(float, ndim=1)

Updated value function vector, of length n.

compute_greedy(*v*, *sigma=None*)
Compute the v-greedy policy.

Parameters **v** : array_like(float, ndim=1)

Value function vector, of length n.

sigma : ndarray(int, ndim=1), optional(default=None)

Optional output array for *sigma*.

Returns **sigma** : ndarray(int, ndim=1)

v-greedy policy vector, of length n.

controlled_mc(*sigma*)
Returns the controlled Markov chain for a given policy *sigma*.

Parameters **sigma** : array_like(int, ndim=1)

Policy vector, of length n.

Returns **mc** : MarkovChain

Controlled Markov chain.

evaluate_policy(*sigma*)
Compute the value of a policy.

Parameters **sigma** : array_like(int, ndim=1)

Policy vector, of length n.

Returns **v_sigma** : ndarray(float, ndim=1)

Value vector of *sigma*, of length n.

modified_policy_iteration(*v_init=None*, *epsilon=None*, *max_iter=None*, *k=20*)
Solve the optimization problem by modified policy iteration. See the *solve* method.

operator_iteration(*T*, *v*, *max_iter*, *tol=None*, **args*, ***kwargs*)
Iteratively apply the operator *T* to *v*. Modify *v* in-place. Iteration is performed for at most a number *max_iter* of times. If *tol* is specified, it is terminated once the distance of *T(v)* from *v* (in the max norm) is less than *tol*.

Parameters **T** : callable

Operator that acts on *v*.

v : ndarray

Object on which T acts. Modified in-place.

max_iter : scalar(int)

Maximum number of iterations.

tol : scalar(float), optional(default=None)

Error tolerance.

args, kwargs :

Other arguments and keyword arguments that are passed directly to the function T each time it is called.

Returns num_iter : scalar(int)

Number of iterations performed.

policy_iteration (*v_init=None*, *max_iter=None*)
Solve the optimization problem by policy iteration. See the *solve* method.

solve (*method='policy_iteration'*, *v_init=None*, *epsilon=None*, *max_iter=None*, *k=20*)
Solve the dynamic programming problem.

Parameters method : str, optinal(default='policy_iteration')

Solution method, str in {'value_iteration', 'vi', 'policy_iteration', 'pi', 'modified_policy_iteration', 'mpi'}.

v_init : array_like(float, ndim=1), optional(default=None)

Initial value function, of length n. If None, *v_init* is set such that v_init(s) = max_a r(s, a) for value iteration and policy iteration; for modified policy iteration, v_init(s) = min_(s_next, a) r(s_next, a)/(1 - beta) to guarantee convergence.

epsilon : scalar(float), optional(default=None)

Value for epsilon-optimality. If None, the value stored in the attribute *epsilon* is used.

max_iter : scalar(int), optional(default=None)

Maximum number of iterations. If None, the value stored in the attribute *max_iter* is used.

k : scalar(int), optional(default=20)

Number of iterations for partial policy evaluation in modified policy iteration (irrelevant for other methods).

Returns res : DPSolveResult

Optimization result represetned as a DPSolveResult. See *DPSolveResult* for details.

value_iteration (*v_init=None*, *epsilon=None*, *max_iter=None*)
Solve the optimization problem by value iteration. See the *solve* method.

`quantecon.markov.ddp.`**backward_induction** (*ddp*, *T*, *v_term=None*)
Solve by backward induction a T-period finite horizon discrete dynamic program with stationary reward and transition probability functions r and q and discount factor $\beta \in [0, 1]$.

The optimal value functions $v_0^*, \dots, v_T^*$ and policy functions $\sigma_0^*, \dots, \sigma_{T-1}^*$ are obtained by $v_T^* = v_T$, and

$$v_{t-1}^*(s) = \max_{a \in A(s)} r(s, a) + \beta \sum_{s' \in S} q(s'|s, a) v_t^*(s') \quad (s \in S)$$

and

$$\sigma^*_{t-1}(s) \in \arg\max_{a \in A(s)} r(s, a) + \beta \sum_{s' \in S} q(s'|s, a) v^*_t(s') \quad (s \in S)$$

for $t = T, \ldots, 1$, where the terminal value function v_T is exogenously given.

Parameters **ddp** : DiscreteDP

DiscreteDP instance storing reward array *R*, transition probability array *Q*, and discount factor *beta*.

T : scalar(int)

Number of decision periods.

v_term : array_like(float, ndim=1), optional(default=None)

Terminal value function, of length equal to n (the number of states). If None, it defaults to the vector of zeros.

Returns **vs** : ndarray(float, ndim=2)

Array of shape (T+1, n) where *vs[t]* contains the optimal value function at period *t = 0, ..., T*.

sigmas : ndarray(int, ndim=2)

Array of shape (T, n) where *sigmas[t]* contains the optimal policy function at period *t = 0, ..., T-1*.

2.4 gth_solve

Filename: gth_solve.py

Author: Daisuke Oyama

Routine to compute the stationary distribution of an irreducible Markov chain by the Grassmann-Taksar-Heyman (GTH) algorithm.

`quantecon.markov.gth_solve.`**`gth_solve`**(*A*, *overwrite=False*, *use_jit=True*)

This routine computes the stationary distribution of an irreducible Markov transition matrix (stochastic matrix) or transition rate matrix (generator matrix) *A*.

More generally, given a Metzler matrix (square matrix whose off-diagonal entries are all nonnegative) *A*, this routine solves for a nonzero solution *x* to *x (A - D) = 0*, where *D* is the diagonal matrix for which the rows of *A - D* sum to zero (i.e., $D_{ii} = \sum_j A_{ij}$ for all i). One (and only one, up to normalization) nonzero solution exists corresponding to each reccurent class of *A*, and in particular, if *A* is irreducible, there is a unique solution; when there are more than one solution, the routine returns the solution that contains in its support the first index *i* such that no path connects *i* to any index larger than *i*. The solution is normalized so that its 1-norm equals one. This routine implements the Grassmann-Taksar-Heyman (GTH) algorithm *[R6]*, a numerically stable variant of Gaussian elimination, where only the off-diagonal entries of *A* are used as the input data. For a nice exposition of the algorithm, see Stewart *[R7]*, Chapter 10.

Parameters **A** : array_like(float, ndim=2)

Stochastic matrix or generator matrix. Must be of shape n x n.

Returns **x** : numpy.ndarray(float, ndim=1)

Stationary distribution of *A*.

overwrite : bool, optional(default=False)

Whether to overwrite *A*.

References

[R6], *[R7]*

2.5 random

Filename: random.py

Author: Daisuke Oyama

Generate MarkovChain and DiscreteDP instances randomly.

`quantecon.markov.random.`**`random_discrete_dp`**(*num_states*, *num_actions*, *beta=None*, *k=None*, *scale=1*, *sparse=False*, *sa_pair=False*, *random_state=None*)

Generate a DiscreteDP randomly. The reward values are drawn from the normal distribution with mean 0 and standard deviation *scale*.

Parameters **num_states** : scalar(int)

Number of states.

num_actions : scalar(int)

Number of actions.

beta : scalar(float), optional(default=None)

Discount factor. Randomly chosen from [0, 1) if not specified.

k : scalar(int), optional(default=None)

Number of possible next states for each state-action pair. Equal to *num_states* if not specified.

scale : scalar(float), optional(default=1)

Standard deviation of the normal distribution for the reward values.

sparse : bool, optional(default=False)

Whether to store the transition probability array in sparse matrix form.

sa_pair : bool, optional(default=False)

Whether to represent the data in the state-action pairs formulation. (If *sparse=True*, automatically set *True*.)

random_state : int or np.random.RandomState, optional

Random seed (integer) or np.random.RandomState instance to set the initial state of the random number generator for reproducibility. If None, a randomly initialized RandomState is used.

Returns **ddp** : DiscreteDP

An instance of DiscreteDP.

quantecon.markov.random.**random_markov_chain**(*n*, *k=None*, *sparse=False*, *random_state=None*)

Return a randomly sampled MarkovChain instance with n states, where each state has k states with positive transition probability.

Parameters **n** : scalar(int)

Number of states.

k : scalar(int), optional(default=None)

Number of states that may be reached from each state with positive probability. Set to n if not specified.

sparse : bool, optional(default=False)

Whether to store the transition probability matrix in sparse matrix form.

random_state : int or np.random.RandomState, optional

Random seed (integer) or np.random.RandomState instance to set the initial state of the random number generator for reproducibility. If None, a randomly initialized RandomState is used.

Returns **mc** : MarkovChain

Examples

```
>>> mc = qe.markov.random_markov_chain(3, random_state=1234)
>>> mc.P
array([[ 0.19151945,  0.43058932,  0.37789123],
       [ 0.43772774,  0.34763084,  0.21464142],
       [ 0.27259261,  0.5073832 ,  0.22002419]])
>>> mc = qe.markov.random_markov_chain(3, k=2, random_state=1234)
>>> mc.P
array([[ 0.19151945,  0.80848055,  0.        ],
       [ 0.        ,  0.62210877,  0.37789123],
       [ 0.56227226,  0.        ,  0.43772774]])
```

quantecon.markov.random.**random_stochastic_matrix**(*n*, *k=None*, *sparse=False*, *format='csr'*, *random_state=None*)

Return a randomly sampled n x n stochastic matrix with k nonzero entries for each row.

Parameters **n** : scalar(int)

Number of states.

k : scalar(int), optional(default=None)

Number of nonzero entries in each row of the matrix. Set to n if not specified.

sparse : bool, optional(default=False)

Whether to generate the matrix in sparse matrix form.

format : str, optional(default='csr')

Sparse matrix format, str in {'bsr', 'csr', 'csc', 'coo', 'lil', 'dia', 'dok'}. Relevant only when sparse=True.

random_state : int or np.random.RandomState, optional

Random seed (integer) or np.random.RandomState instance to set the initial state of the random number generator for reproducibility. If None, a randomly initialized RandomState is used.

Returns P : numpy ndarray or scipy sparse matrix (float, ndim=2)

Stochastic matrix.

See also:

random_markov_chain Return a random MarkovChain instance.

2.6 utilities

Utility routines for the markov submodule

quantecon.markov.utilities.**sa_indices**

Generate *s_indices* and *a_indices* for *DiscreteDP*, for the case where all the actions are feasible at every state.

Parameters num_states : scalar(int)

Number of states.

num_actions : scalar(int)

Number of actions.

Returns s_indices : ndarray(int, ndim=1)

Array containing the state indices.

a_indices : ndarray(int, ndim=1)

Array containing the action indices.

Examples

```
>>> s_indices, a_indices = qe.markov.sa_indices(4, 3)
>>> s_indices
array([0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3])
>>> a_indices
array([0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2])
```

CHAPTER 3

Random

3.1 utilities

Utilities to Support Random Operations and Generating Vectors and Matrices

quantecon.random.utilities.**probvec**(*m*, *k*, *random_state=None*, *parallel=True*)
Return m randomly sampled probability vectors of dimension k.

Parameters **m** : scalar(int)

Number of probability vectors.

k : scalar(int)

Dimension of each probability vectors.

random_state : int or np.random.RandomState, optional

Random seed (integer) or np.random.RandomState instance to set the initial state of the random number generator for reproducibility. If None, a randomly initialized RandomState is used.

parallel : bool(default=True)

Whether to use multi-core CPU (parallel=True) or single-threaded CPU (parallel=False). (Internally the code is executed through Numba.guvectorize.)

Returns **x** : ndarray(float, ndim=2)

Array of shape (m, k) containing probability vectors as rows.

Examples

```
>>> qe.random.probvec(2, 3, random_state=1234)
array([[ 0.19151945,  0.43058932,  0.37789123],
       [ 0.43772774,  0.34763084,  0.21464142]])
```

quantecon.random.utilities.**sample_without_replacement**

Randomly choose k integers without replacement from 0, ..., n-1.

Parameters **n** : scalar(int)

Number of integers, 0, ..., n-1, to sample from.

k : scalar(int)

Number of integers to sample.

num_trials : scalar(int), optional(default=None)

Number of trials.

random_state : int or np.random.RandomState, optional

Random seed (integer) or np.random.RandomState instance to set the initial state of the random number generator for reproducibility. If None, a randomly initialized RandomState is used.

Returns **result** : ndarray(int, ndim=1 or 2)

Array of shape (k,) if num_trials is None, or of shape (num_trials, k) otherwise, (each row of) which contains k unique random elements chosen from 0, ..., n-1.

Examples

```
>>> qe.random.sample_without_replacement(5, 3, random_state=1234)
array([0, 2, 1])
>>> qe.random.sample_without_replacement(5, 3, num_trials=4,
...                                     random_state=1234)
array([[0, 2, 1],
       [3, 4, 0],
       [1, 3, 2],
       [4, 1, 3]])
```

CHAPTER 4

Tools

4.1 arma

Filename: arma.py Authors: Doc-Jin Jang, Jerry Choi, Thomas Sargent, John Stachurski

Provides functions for working with and visualizing scalar ARMA processes.

TODO: 1. Fix warnings concerning casting complex variables back to floats

class `quantecon.arma.`**`ARMA`**(*phi*, *theta=0*, *sigma=1*)

Bases: `object`

This class represents scalar ARMA(p, q) processes.

If phi and theta are scalars, then the model is understood to be

$$X_t = \phi X_{t-1} + \epsilon_t + \theta \epsilon_{t-1}$$

where $epsilon_t$ is a white noise process with standard deviation $sigma$. If phi and theta are arrays or sequences, then the interpretation is the ARMA(p, q) model

$$\begin{aligned} X_t = \phi_1 X_{t-1} + ... + \phi_p X_{t-p} + \\ \epsilon_t + \theta_1 \epsilon_{t-1} + ... + \theta_q \epsilon_{t-q} \end{aligned}$$

where

- $\phi = (\phi_1, \phi_2, ..., \phi_p)$
- $\theta = (\theta_1, \theta_2, ..., \theta_q)$
- σ is a scalar, the standard deviation of the white noise

Parameters **phi** : scalar or iterable or array_like(float)

Autocorrelation values for the autocorrelated variable. See above for explanation.

theta : scalar or iterable or array_like(float)

Autocorrelation values for the white noise of the model. See above for explanation

sigma : scalar(float)

The standard deviation of the white noise

Attributes

phi, theta, sigma	(see Parmeters)
ar_poly	(array_like(float)) The polynomial form that is needed by scipy.signal to do the processing we desire. Corresponds with the phi values
ma_poly	(array_like(float)) The polynomial form that is needed by scipy.signal to do the processing we desire. Corresponds with the theta values

Methods

autocovariance([num_autocov])	Compute the autocovariance function from the ARMA parameters over the integers range(num_autocov) using the spectral density and the inverse Fourier transform.
impulse_response([impulse_length])	Get the impulse response corresponding to our model.
plot_autocovariance([ax, show])	
plot_impulse_response([ax, show])	
plot_simulation([ax, show])	
plot_spectral_density([ax, show])	
quad_plot()	Plots the impulse response, spectral_density, autocovariance, and one realization of the process.
set_params()	Internally, scipy.signal works with systems of the form
simulation([ts_length])	Compute a simulated sample path assuming Gaussian shocks.
spectral_density([two_pi, res])	Compute the spectral density function.

autocovariance(*num_autocov=16*)

Compute the autocovariance function from the ARMA parameters over the integers range(num_autocov) using the spectral density and the inverse Fourier transform.

Parameters num_autocov : scalar(int), optional(default=16)

The number of autocovariances to calculate

impulse_response(*impulse_length=30*)

Get the impulse response corresponding to our model.

Returns psi : array_like(float)

psi[j] is the response at lag j of the impulse response. We take psi[0] as unity.

phi

plot_autocovariance(*ax=None, show=True*)

plot_impulse_response(*ax=None, show=True*)

plot_simulation(*ax=None, show=True*)

plot_spectral_density(*ax=None, show=True*)

quad_plot()

Plots the impulse response, spectral_density, autocovariance, and one realization of the process.

set_params()

Internally, scipy.signal works with systems of the form

$$ar_{poly}(L)X_t = ma_{poly}(L)\epsilon_t$$

where L is the lag operator. To match this, we set

$$ar_{poly} = (1, -\phi_1, -\phi_2, ..., -\phi_p)$$
$$ma_{poly} = (1, \theta_1, \theta_2, ..., \theta_q)$$

In addition, ar_poly must be at least as long as ma_poly. This can be achieved by padding it out with zeros when required.

simulation(*ts_length=90*)

Compute a simulated sample path assuming Gaussian shocks.

Parameters **ts_length** : scalar(int), optional(default=90)

Number of periods to simulate for

Returns **vals** : array_like(float)

A simulation of the model that corresponds to this class

spectral_density(*two_pi=True*, *res=1200*)

Compute the spectral density function. The spectral density is the discrete time Fourier transform of the autocovariance function. In particular,

$$f(w) = \sum_k \gamma(k) exp(-ikw)$$

where gamma is the autocovariance function and the sum is over the set of all integers.

Parameters **two_pi** : Boolean, optional

Compute the spectral density function over [0, pi] if two_pi is False and [0, 2 pi] otherwise. Default value is True

res : scalar or array_like(int), optional(default=1200)

If res is a scalar then the spectral density is computed at *res* frequencies evenly spaced around the unit circle, but if res is an array then the function computes the response at the frequencies given by the array

Returns **w** : array_like(float)

The normalized frequencies at which h was computed, in radians/sample

spect : array_like(float)

The frequency response

theta

4.2 cartesian

Filename: cartesian.py

Authors: Pablo Winant

Implements cartesian products and regular cartesian grids.

`quantecon.cartesian.`**`cartesian`**`(`*nodes*, *order='C'*`)`

Cartesian product of a list of arrays

Parameters nodes: (list of 1d-arrays)

order: ('C' or 'F') order in which the product is enumerated

Returns out: (2d-array) each line corresponds to one point of the product space

`quantecon.cartesian.`**`mlinspace`**`(`*a*, *b*, *nums*, *order='C'*`)`

Constructs a regular cartesian grid

Parameters a: (1d-array) lower bounds in each dimension

b: (1d-array) upper bounds in each dimension

nums: (1d-array) number of nodes along each dimension

order: ('C' or 'F') order in which the product is enumerated

Returns out: (2d-array) each line corresponds to one point of the product space

4.3 ce_util

Filename: ce_util.py Authors: Chase Coleman, Spencer Lyon, John Stachurski, and Thomas Sargent Date: 2014-07-01

Utility functions used in CompEcon

Based routines found in the CompEcon toolbox by Miranda and Fackler.

4.3.1 References

Miranda, Mario J, and Paul L Fackler. Applied Computational Economics and Finance, MIT Press, 2002.

`quantecon.ce_util.`**`ckron`**`(`**arrays*`)`

Repeatedly applies the np.kron function to an arbitrary number of input arrays

Parameters *arrays : tuple/list of np.ndarray

Returns out : np.ndarray

The result of repeated kronecker products

Notes

Based of original function *ckron* in CompEcon toolbox by Miranda and Fackler

References

Miranda, Mario J, and Paul L Fackler. Applied Computational Economics and Finance, MIT Press, 2002.

`quantecon.ce_util.`**`gridmake`**`(`**arrays*`)`

TODO: finish this docstring

Notes

Based of original function `gridmake` in CompEcon toolbox by Miranda and Fackler

References

Miranda, Mario J, and Paul L Fackler. Applied Computational Economics and Finance, MIT Press, 2002.

4.4 compute_fp

Filename: compute_fp.py Authors: Thomas Sargent, John Stachurski, Daisuke Oyama

Compute an approximate fixed point of a given operator T, starting from specified initial condition v.

`quantecon.compute_fp.`**`compute_fixed_point`**(*T*, *v*, *error_tol=0.001*, *max_iter=50*, *verbose=2*, *print_skip=5*, *method='iteration'*, **args*, ***kwargs*)

Computes and returns an approximate fixed point of the function *T*.

The default method *'iteration'* simply iterates the function given an initial condition *v* and returns $T^k v$ when the condition $\|T^k v - T^{k-1} v\| \leq \mathrm{error\_tol}$ is satisfied or the number of iterations k reaches *max_iter*. Provided that *T* is a contraction mapping or similar, $T^k v$ will be an approximation to the fixed point.

The method *'imitation_game'* uses the "imitation game algorithm" developed by McLennan and Tourky *[R8]*, which internally constructs a sequence of two-player games called imitation games and utilizes their Nash equilibria, computed by the Lemke-Howson algorithm routine. It finds an approximate fixed point of *T*, a point v^* such that $\|T(v) - v\| \leq \mathrm{error\_tol}$, provided *T* is a function that satisfies the assumptions of Brouwer's fixed point theorm, i.e., a continuous function that maps a compact and convex set to itself.

Parameters **T** : callable

A callable object (e.g., function) that acts on v

v : object

An object such that T(v) is defined; modified in place if *method='iteration' and 'v* is an array

error_tol : scalar(float), optional(default=1e-3)

Error tolerance

max_iter : scalar(int), optional(default=50)

Maximum number of iterations

verbose : scalar(int), optional(default=2)

Level of feedback (0 for no output, 1 for warnings only, 2 for warning and residual error reports during iteration)

print_skip : scalar(int), optional(default=5)

How many iterations to apply between print messages (effective only when *verbose=2*)

method : str, optional(default='iteration')

str in {'iteration', 'imitation_game'}. Method of computing an approximate fixed point

args, kwargs :

Other arguments and keyword arguments that are passed directly to the function T each time it is called

Returns v : object

The approximate fixed point

References

[R8]

4.5 discrete_rv

Filename: discrete_rv.py

Authors: Thomas Sargent, John Stachurski

Generates an array of draws from a discrete random variable with a specified vector of probabilities.

class `quantecon.discrete_rv.`**`DiscreteRV`**(*q*)

Bases: `object`

Generates an array of draws from a discrete random variable with vector of probabilities given by q.

Parameters q : array_like(float)

Nonnegative numbers that sum to 1

Attributes

q	Getter method for q.

Q	(array_like(float)) The cumulative sum of q

Methods

draw([k])	Returns k draws from q.

`draw`(*k=1*)

Returns k draws from q.

For each such draw, the value i is returned with probability q[i].

Parameters k : scalar(int), optional

Number of draws to be returned

Returns array_like(int)

An array of k independent draws from q

`q`

Getter method for q.

`quantecon.discrete_rv.`**`uniform`**(*low=0.0*, *high=1.0*, *size=None*)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

Parameters **low** : float or array_like of floats, optional

Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high : float or array_like of floats

Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `low` and `high` are both scalars. Otherwise, `np.broadcast(low, high).size` samples are drawn.

Returns **out** : ndarray or scalar

Drawn samples from the parameterized uniform distribution.

See also:

`randint` Discrete uniform distribution, yielding integers.

`random_integers` Discrete uniform distribution over the closed interval `[low, high]`.

`random_sample` Floats uniformly distributed over `[0, 1)`.

`random` Alias for *random_sample*.

`rand` Convenience function that accepts dimensions as input, e.g., `rand(2,2)` would generate a 2-by-2 array of floats, uniformly distributed over `[0, 1)`.

Notes

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b - a}$$

anywhere within the interval `[a, b)`, and zero elsewhere.

When `high` == `low`, values of `low` will be returned. If `high` < `low`, the results are officially undefined and may eventually raise an error, i.e. do not rely on this function to behave when passed arguments satisfying that inequality condition.

Examples

Draw samples from the distribution:

```
>>> s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

4.6 distributions

Filename: distributions.py

Probability distributions useful in economics.

4.6.1 References

http://en.wikipedia.org/wiki/Beta-binomial_distribution

class `quantecon.distributions.BetaBinomial`(*n*, *a*, *b*)

Bases: `object`

The Beta-Binomial distribution

Parameters **n** : scalar(int)

First parameter to the Beta-binomial distribution

a : scalar(float)

Second parameter to the Beta-binomial distribution

b : scalar(float)

Third parameter to the Beta-binomial distribution

Attributes

n, a, b	(see Parameters)

Methods

pdf()	Generate the vector of probabilities for the Beta-binomial (n, a, b) distribution.

mean

pdf()

Generate the vector of probabilities for the Beta-binomial (n, a, b) distribution.

The Beta-binomial distribution takes the form

$$p(k \mid n, a, b) = \binom{n}{k} \frac{B(k+a, n-k+b)}{B(a,b)}, \qquad k = 0, \ldots, n,$$

where B is the beta function.

Parameters **n** : scalar(int)

First parameter to the Beta-binomial distribution

a : scalar(float)

Second parameter to the Beta-binomial distribution

b : scalar(float)

Third parameter to the Beta-binomial distribution

Returns probs: array_like(float)

Vector of probabilities over k

skew
skewness

std
standard deviation

var
Variance

4.7 ecdf

Filename: ecdf.py

Authors: Thomas Sargent, John Stachurski

Implements the empirical cumulative distribution function given an array of observations.

class `quantecon.ecdf.`**`ECDF`**(*observations*)
Bases: `object`

One-dimensional empirical distribution function given a vector of observations.

Parameters **observations** : array_like

An array of observations

Attributes

observations	(see Parameters)

Methods

`__call__`(x)	Evaluates the ecdf at x

4.8 estspec

Filename: estspec.py

Authors: Thomas Sargent, John Stachurski

Functions for working with periodograms of scalar data.

`quantecon.estspec.ar_periodogram(x, window='hanning', window_len=7)`

Compute periodogram from data x, using prewhitening, smoothing and recoloring. The data is fitted to an AR(1) model for prewhitening, and the residuals are used to compute a first-pass periodogram with smoothing. The fitted coefficients are then used for recoloring.

Parameters **x** : array_like(float)

A flat NumPy array containing the data to smooth

window_len : scalar(int), optional

An odd integer giving the length of the window. Defaults to 7.

window : string

A string giving the window type. Possible values are 'flat', 'hanning', 'hamming', 'bartlett' or 'blackman'

Returns **w** : array_like(float)

Fourier frequences at which periodogram is evaluated

I_w : array_like(float)

Values of periodogram at the Fourier frequences

`quantecon.estspec.periodogram(x, window=None, window_len=7)`

Computes the periodogram

$$I(w) = (1/n)|sum_{t=0}^{n-1} x_t e^{itw}|^2$$

at the Fourier frequences w_j := 2 pi j / n, j = 0, ..., n - 1, using the fast Fourier transform. Only the frequences w_j in [0, pi] and corresponding values I(w_j) are returned. If a window type is given then smoothing is performed.

Parameters **x** : array_like(float)

A flat NumPy array containing the data to smooth

window_len : scalar(int), optional(default=7)

An odd integer giving the length of the window. Defaults to 7.

window : string

A string giving the window type. Possible values are 'flat', 'hanning', 'hamming', 'bartlett' or 'blackman'

Returns **w** : array_like(float)

Fourier frequences at which periodogram is evaluated

I_w : array_like(float)

Values of periodogram at the Fourier frequences

`quantecon.estspec.smooth(x, window_len=7, window='hanning')`

Smooth the data in x using convolution with a window of requested size and type.

Parameters **x** : array_like(float)

A flat NumPy array containing the data to smooth

window_len : scalar(int), optional

An odd integer giving the length of the window. Defaults to 7.

window : string

A string giving the window type. Possible values are 'flat', 'hanning', 'hamming', 'bartlett' or 'blackman'

Returns array_like(float)

The smoothed values

Notes

Application of the smoothing window at the top and bottom of x is done by reflecting x around these points to extend it sufficiently in each direction.

4.9 graph_tools

Filename: graph_tools.py

Author: Daisuke Oyama

Tools for dealing with a directed graph.

class `quantecon.graph_tools.DiGraph` (*adj_matrix*, *weighted=False*, *node_labels=None*)

Bases: `object`

Class for a directed graph. It stores useful information about the graph structure such as strong connectivity *[R9]* and periodicity *[R10]*.

Parameters **adj_matrix** : array_like(ndim=2)

Adjacency matrix representing a directed graph. Must be of shape n x n.

weighted : bool, optional(default=False)

Whether to treat *adj_matrix* as a weighted adjacency matrix.

node_labels : array_like(default=None)

Array_like of length n containing the labels associated with the nodes, which must be homogeneous in type. If None, the labels default to integers 0 through n-1.

References

[R9], *[R10]*

Attributes

csgraph	(scipy.sparse.csr_matrix) Compressed sparse representation of the digraph.
is_strongly_connected	(bool) Indicate whether the digraph is strongly connected.
num_strongly_connected_components	(int) The number of the strongly connected components.
strongly_connected_components_indices	(list(ndarray(int))) List of numpy arrays containing the indices of the strongly connected components.
strongly_connected_components	(list(ndarray)) List of numpy arrays containing the strongly connected components, where the nodes are annotated with their labels (if *node_labels* is not None).
num_sink_strongly_connected_components	(int) The number of the sink strongly connected components.
sink_strongly_connected_components_indices	(list(ndarray(int))) List of numpy arrays containing the indices of the sink strongly connected components.
sink_strongly_connected_components	(list(ndarray)) List of numpy arrays containing the sink strongly connected components, where the nodes are annotated with their labels (if *node_labels* is not None).
is_aperiodic	(bool) Indicate whether the digraph is aperiodic.
period	(int) The period of the digraph. Defined only for a strongly connected digraph.
cyclic_components_indices	(list(ndarray(int))) List of numpy arrays containing the indices of the cyclic components.
cyclic_components	(list(ndarray)) List of numpy arrays containing the cyclic components, where the nodes are annotated with their labels (if *node_labels* is not None).

Methods

`subgraph`(nodes)	Return the subgraph consisting of the given nodes and edges between thses nodes.

cyclic_components

cyclic_components_indices

is_aperiodic

is_strongly_connected

node_labels

num_sink_strongly_connected_components

num_strongly_connected_components

period

scc_proj

sink_scc_labels

sink_strongly_connected_components

sink_strongly_connected_components_indices

strongly_connected_components

strongly_connected_components_indices

subgraph(*nodes*)

Return the subgraph consisting of the given nodes and edges between thses nodes.

Parameters **nodes** : array_like(int, ndim=1)

Array of node indices.

Returns DiGraph

A DiGraph representing the subgraph.

`quantecon.graph_tools.`**`annotate_nodes`**(*func*)

4.10 gridtools

Filename: gridtools.py

Authors: Pablo Winant

Implements cartesian products and regular cartesian grids.

`quantecon.gridtools.`**`cartesian`**(*nodes*, *order='C'*)

Cartesian product of a list of arrays

Parameters **nodes: (list of 1d-arrays)**

order: ('C' or 'F') order in which the product is enumerated

Returns out: (2d-array) each line corresponds to one point of the product space

`quantecon.gridtools.`**`mlinspace`**(*a*, *b*, *nums*, *order='C'*)

Constructs a regular cartesian grid

Parameters **a: (1d-array) lower bounds in each dimension**

b: (1d-array) upper bounds in each dimension

nums: (1d-array) number of nodes along each dimension

order: ('C' or 'F') order in which the product is enumerated

Returns out: (2d-array) each line corresponds to one point of the product space

4.11 ivp

Base class for solving initial value problems (IVPs) of the form:

$$\frac{dy}{dt} = f(t, y),\ y(t_0) = y_0$$

using finite difference methods. The *quantecon.ivp* class uses various integrators from the *scipy.integrate.ode* module to perform the integration (i.e., solve the ODE) and parametric B-spline interpolation from *scipy.interpolate* to approximate the value of the solution between grid points. The *quantecon.ivp* module also provides a method for computing the residual of the solution which can be used for assessing the overall accuracy of the approximated solution.

@author : David R. Pugh @date : 2014-09-09

class `quantecon.ivp.`**`IVP`**(*f*, *jac=None*)

Bases: `scipy.integrate._ode.ode`

Attributes

y

Methods

compute_residual(traj, ti[, k, ext])	The residual is the difference between the derivative of the B-spline approximation of the solution trajectory and the right-hand side of the original ODE evaluated along the approximated solution trajectory.
integrate(t[, step, relax])	Find y=y(t), set y as an initial condition, and return y.
interpolate(traj, ti[, k, der, ext])	Parametric B-spline interpolation in N-dimensions.
set_f_params(*args)	Set extra parameters for user-supplied function f.
set_initial_value(y[, t])	Set initial conditions y(t) = y.
set_integrator(name, **integrator_params)	Set integrator by name.
set_jac_params(*args)	Set extra parameters for user-supplied function jac.
set_solout(solout)	Set callable to be called at every successful integration step.
solve(t0, y0[, h, T, g, tol, integrator, ...])	Solve the IVP by integrating the ODE given some initial condition.
successful()	Check if integration was successful.

compute_residual(*traj*, *ti*, *k=3*, *ext=2*)

The residual is the difference between the derivative of the B-spline approximation of the solution trajectory and the right-hand side of the original ODE evaluated along the approximated solution trajectory.

Parameters **traj** : array_like (float)

Solution trajectory providing the data points for constructing the B-spline representation.

ti : array_like (float)

Array of values for the independent variable at which to interpolate the value of the B-spline.

k : int, optional(default=3)

Degree of the desired B-spline. Degree must satisfy $1 \le k \le 5$.

ext : int, optional(default=2)

Controls the value of returned elements for outside the original knot sequence provided by traj. For extrapolation, set *ext=0*; *ext=1* returns zero; *ext=2* raises a *ValueError*.

Returns **residual** : array (float)

Difference between the derivative of the B-spline approximation of the solution trajectory and the right-hand side of the ODE evaluated along the approximated solution trajectory.

interpolate(*traj*, *ti*, *k=3*, *der=0*, *ext=2*)

Parametric B-spline interpolation in N-dimensions.

Parameters **traj** : array_like (float)

Solution trajectory providing the data points for constructing the B-spline representation.

ti : array_like (float)

Array of values for the independent variable at which to interpolate the value of the B-spline.

k : int, optional(default=3)

Degree of the desired B-spline. Degree must satisfy $1 \leq k \leq 5$.

der : int, optional(default=0)

The order of derivative of the spline to compute (must be less than or equal to *k*).

ext : int, optional(default=2) Controls the value of returned elements

for outside the original knot sequence provided by traj. For extrapolation, set *ext=0*; *ext=1* returns zero; *ext=2* raises a *ValueError*.

Returns interp_traj: ndarray (float)

The interpolated trajectory.

solve(*t0*, *y0*, *h=1.0*, *T=None*, *g=None*, *tol=None*, *integrator='dopri5'*, *step=False*, *relax=False*, ***kwargs*)

Solve the IVP by integrating the ODE given some initial condition.

Parameters **t0** : float

Initial condition for the independent variable.

y0 : array_like (float, shape=(n,))

Initial condition for the dependent variables.

h : float, optional(default=1.0)

Step-size for computing the solution. Can be positive or negative depending on the desired direction of integration.

T : int, optional(default=None)

Terminal value for the independent variable. One of either *T* or *g* must be specified.

g : callable `g(t, y, f_args)`, optional(default=None)

Provides a stopping condition for the integration. If specified user must also specify a stopping tolerance, *tol*.

tol : float, optional (default=None)

Stopping tolerance for the integration. Only required if *g* is also specifed.

integrator : str, optional(default='dopri5')

Must be one of 'vode', 'lsoda', 'dopri5', or 'dop853'

step : bool, optional(default=False)

Allows access to internal steps for those solvers that use adaptive step size routines. Currently only 'vode', 'zvode', and 'lsoda' support *step=True*.

relax : bool, optional(default=False)

Currently only 'vode', 'zvode', and 'lsoda' support *relax=True*.

****kwargs** : dict, optional(default=None)

Dictionary of integrator specific keyword arguments. See the Notes section of the docstring for *scipy.integrate.ode* for a complete description of solver specific keyword arguments.

Returns solution: ndarray (float)

Simulated solution trajectory.

4.12 kalman

Filename: kalman.py Reference: http://quant-econ.net/py/kalman.html

Implements the Kalman filter for a linear Gaussian state space model.

class `quantecon.kalman.`**`Kalman`**(*ss*, *x_hat=None*, *Sigma=None*)

Bases: `object`

Implements the Kalman filter for the Gaussian state space model

x_{t+1} = A x_t + C w_{t+1} y_t = G x_t + H v_t.

Here x_t is the hidden state and y_t is the measurement. The shocks w_t and v_t are iid standard normals. Below we use the notation

Q := CC' R := HH'

Parameters **ss** : instance of LinearStateSpace

An instance of the quantecon.lss.LinearStateSpace class

x_hat : scalar(float) or array_like(float), optional(default=None)

An n x 1 array representing the mean x_hat and covariance matrix Sigma of the prior/predictive density. Set to zero if not supplied.

Sigma : scalar(float) or array_like(float), optional(default=None)

An n x n array representing the covariance matrix Sigma of the prior/predictive density. Must be positive definite. Set to the identity if not supplied.

References

http://quant-econ.net/py/kalman.html

Attributes

Sigma, x_hat	(as above)
Sigma_infinity	(array_like or scalar(float)) The infinite limit of Sigma_t
K_infinity	(array_like or scalar(float)) The stationary Kalman gain.

Methods

filtered_to_forecast()	Updates the moments of the time t filtering distribution to the

Continued on next page

Table 4.9 – continued from previous page

`prior_to_filtered`(y)	Updates the moments (x_hat, Sigma) of the time t prior to the time t filtering distribution, using current measurement y_t.
`set_state`(x_hat, Sigma)	
`stationary_coefficients`(j[, coeff_type])	Wold representation moving average or VAR coefficients for the steady state Kalman filter.
`stationary_innovation_covar`()	
`stationary_values`()	Computes the limit of Sigma_t as t goes to infinity by solving the associated Riccati equation.
`update`(y)	Updates x_hat and Sigma given k x 1 ndarray y.
`whitener_lss`()	This function takes the linear state space system

filtered_to_forecast()

Updates the moments of the time t filtering distribution to the moments of the predictive distribution, which becomes the time t+1 prior

prior_to_filtered(*y*)

Updates the moments (x_hat, Sigma) of the time t prior to the time t filtering distribution, using current measurement y_t.

The updates are according to

x_{hat}^F = x_{hat} + Sigma G' (G Sigma G' + R)^{-1} (y - G x_{hat})

Sigma^F = Sigma - Sigma G' (G Sigma G' + R)^{-1} G Sigma

Parameters **y** : scalar or array_like(float)

The current measurement

set_state(*x_hat*, *Sigma*)

stationary_coefficients(*j*, *coeff_type='ma'*)

Wold representation moving average or VAR coefficients for the steady state Kalman filter.

Parameters **j** : int

The lag length

coeff_type : string, either 'ma' or 'var' (default='ma')

The type of coefficent sequence to compute. Either 'ma' for moving average or 'var' for VAR.

stationary_innovation_covar()

stationary_values()

Computes the limit of Sigma_t as t goes to infinity by solving the associated Riccati equation. Computation is via the doubling algorithm (see the documentation in *matrix_eqn.solve_discrete_riccati*).

Returns **Sigma_infinity** : array_like or scalar(float)

The infinite limit of Sigma_t

K_infinity : array_like or scalar(float)

The stationary Kalman gain.

update(*y*)

Updates x_hat and Sigma given k x 1 ndarray y. The full update, from one period to the next

Parameters **y** : np.ndarray

A k x 1 ndarray y representing the current measurement

whitener_lss()

This function takes the linear state space system that is an input to the Kalman class and it converts that system to the time-invariant whitener represenation given by

tilde{x}_{t+1}^* = tilde{A} tilde{x} + tilde{C} v a = tilde{G} tilde{x}

where

tilde{x}_t = [x+{t}, hat{x}_{t}, v_{t}]

and

tilde{A} = [A 0 0 KG A-KG KH 0 0 0]

tilde{C} = [C 0 0 0 0 I]

tilde{G} = [G -G H]

with A, C, G, H coming from the linear state space system that defines the Kalman instance

Returns **whitened_lss** : LinearStateSpace

This is the linear state space system that represents the whitened system

4.13 lae

Filename: lae.py

Authors: Thomas J. Sargent, John Stachurski,

Computes a sequence of marginal densities for a continuous state space Markov chain X_t where the transition probabilities can be represented as densities. The estimate of the marginal density of X_t is

rac{1}{n} sum_{i=0}^n p(X_{t-1}^i, y)

This is a density in y.

4.13.1 References

http://quant-econ.net/py/stationary_densities.html

class quantecon.lae.**LAE**(*p*, *X*)

Bases: object

An instance is a representation of a look ahead estimator associated with a given stochastic kernel p and a vector of observations X.

Parameters **p** : function

The stochastic kernel. A function p(x, y) that is vectorized in both x and y

X : array_like(float)

A vector containing observations

Examples

```
>>> psi = LAE(p, X)
>>> y = np.linspace(0, 1, 100)
>>> psi(y)  # Evaluate look ahead estimate at grid of points y
```

Attributes

p, X	(see Parameters)

Methods

`__call__`(y)	A vectorized function that returns the value of the look ahead estimate at the values in the array y.

4.14 lqcontrol

Filename: lqcontrol.py

Authors: Thomas J. Sargent, John Stachurski

Provides a class called LQ for solving linear quadratic control problems.

class `quantecon.lqcontrol.`**`LQ`**(*Q*, *R*, *A*, *B*, *C=None*, *N=None*, *beta=1*, *T=None*, *Rf=None*)
Bases: `object`

This class is for analyzing linear quadratic optimal control problems of either the infinite horizon form

. min E sum_{t=0}^{infty} beta^t r(x_t, u_t)

with

r(x_t, u_t) := x_t' R x_t + u_t' Q u_t + 2 u_t' N x_t

or the finite horizon form

min E sum_{t=0}^{T-1} beta^t r(x_t, u_t) + beta^T x_T' R_f x_T

Both are minimized subject to the law of motion

x_{t+1} = A x_t + B u_t + C w_{t+1}

Here x is n x 1, u is k x 1, w is j x 1 and the matrices are conformable for these dimensions. The sequence {w_t} is assumed to be white noise, with zero mean and E w_t w_t = I, the j x j identity.

If C is not supplied as a parameter, the model is assumed to be deterministic (and C is set to a zero matrix of appropriate dimension).

For this model, the time t value (i.e., cost-to-go) function V_t takes the form

x' P_T x + d_T

and the optimal policy is of the form u_T = -F_T x_T. In the infinite horizon case, V, P, d and F are all stationary.

Parameters Q : array_like(float)

Q is the payoff(or cost) matrix that corresponds with the control variable u and is k x k. Should be symmetric and nonnegative definite

R : array_like(float)

R is the payoff(or cost) matrix that corresponds with the state variable x and is n x n. Should be symetric and non-negative definite

N : array_like(float)

N is the cross product term in the payoff, as above. It should be k x n.

A : array_like(float)

A is part of the state transition as described above. It should be n x n

B : array_like(float)

B is part of the state transition as described above. It should be n x k

C : array_like(float), optional(default=None)

C is part of the state transition as described above and corresponds to the random variable today. If the model is deterministic then C should take default value of None

beta : scalar(float), optional(default=1)

beta is the discount parameter

T : scalar(int), optional(default=None)

T is the number of periods in a finite horizon problem.

Rf : array_like(float), optional(default=None)

Rf is the final (in a finite horizon model) payoff(or cost) matrix that corresponds with the control variable u and is n x n. Should be symetric and non-negative definite

Attributes

Q, R, N, A, B, C, beta, T, Rf	(see Parameters)
P	(array_like(float)) P is part of the value function representation of V(x) = x'Px + d
d	(array_like(float)) d is part of the value function representation of V(x) = x'Px + d
F	(array_like(float)) F is the policy rule that determines the choice of control in each period.
k, n, j	(scalar(int)) The dimensions of the matrices as presented above

Methods

`compute_sequence`(x0[, ts_length])	Compute and return the optimal state and control sequences x_0, ..., x_T and u_0,..., u_T under the assumption that {w_t} is iid and N(0, 1).
`stationary_values`()	Computes the matrix P and scalar d that represent the value
`update_values`()	This method is for updating in the finite horizon case.

compute_sequence (*x0*, *ts_length=None*)

Compute and return the optimal state and control sequences x_0, ..., x_T and u_0,..., u_T under the assumption that {w_t} is iid and N(0, 1).

Parameters **x0** : array_like(float)

The initial state, a vector of length n

ts_length : scalar(int)

Length of the simulation – defaults to T in finite case

Returns **x_path** : array_like(float)

An n x T+1 matrix, where the t-th column represents x_t

u_path : array_like(float)

A k x T matrix, where the t-th column represents u_t

w_path : array_like(float)

A j x T+1 matrix, where the t-th column represent w_t

stationary_values ()

Computes the matrix P and scalar d that represent the value function

V(x) = x' P x + d

in the infinite horizon case. Also computes the control matrix F from u = - Fx

Returns **P** : array_like(float)

P is part of the value function representation of V(x) = xPx + d

F : array_like(float)

F is the policy rule that determines the choice of control in each period.

d : array_like(float)

d is part of the value function representation of V(x) = xPx + d

update_values ()

This method is for updating in the finite horizon case. It shifts the current value function

V_t(x) = x' P_t x + d_t

and the optimal policy F_t one step *back* in time, replacing the pair P_t and d_t with P_{t-1} and d_{t-1}, and F_t with F_{t-1}

4.15 lqnash

quantecon.lqnash.**nnash** (*A*, *B1*, *B2*, *R1*, *R2*, *Q1*, *Q2*, *S1*, *S2*, *W1*, *W2*, *M1*, *M2*, *beta=1.0*, *tol=1e-08*, *max_iter=1000*)

Compute the limit of a Nash linear quadratic dynamic game. In this problem, player i minimizes

$$\sum_{t=0}^{\infty} \{x_t' r_i x_t + 2x_t' w_i u_{it} + u_{it}' q_i u_{it} + u_{jt}' s_i u_{jt} + 2u_{jt}' m_i u_{it}\}$$

subject to the law of motion

$$x_{t+1} = Ax_t + b_1 u_{1t} + b_2 u_{2t}$$

and a perceived control law $u_j(t) = -f_j x_t$ for the other player.

The solution computed in this routine is the f_i and p_i of the associated double optimal linear regulator problem.

Parameters **A** : scalar(float) or array_like(float)

Corresponds to the above equation, should be of size (n, n)

B1 : scalar(float) or array_like(float)

As above, size (n, k_1)

B2 : scalar(float) or array_like(float)

As above, size (n, k_2)

R1 : scalar(float) or array_like(float)

As above, size (n, n)

R2 : scalar(float) or array_like(float)

As above, size (n, n)

Q1 : scalar(float) or array_like(float)

As above, size (k_1, k_1)

Q2 : scalar(float) or array_like(float)

As above, size (k_2, k_2)

S1 : scalar(float) or array_like(float)

As above, size (k_1, k_1)

S2 : scalar(float) or array_like(float)

As above, size (k_2, k_2)

W1 : scalar(float) or array_like(float)

As above, size (n, k_1)

W2 : scalar(float) or array_like(float)

As above, size (n, k_2)

M1 : scalar(float) or array_like(float)

As above, size (k_2, k_1)

M2 : scalar(float) or array_like(float)

As above, size (k_1, k_2)

beta : scalar(float), optional(default=1.0)

Discount rate

tol : scalar(float), optional(default=1e-8)

This is the tolerance level for convergence

max_iter : scalar(int), optional(default=1000)

This is the maximum number of iteratiosn allowed

Returns **F1** : array_like, dtype=float, shape=(k_1, n)

Feedback law for agent 1

F2 : array_like, dtype=float, shape=(k_2, n)

Feedback law for agent 2

P1 : array_like, dtype=float, shape=(n, n)

The steady-state solution to the associated discrete matrix Riccati equation for agent 1

P2 : array_like, dtype=float, shape=(n, n)

The steady-state solution to the associated discrete matrix Riccati equation for agent 2

4.16 lss

Filename: lss.py Reference: http://quant-econ.net/py/linear_models.html

Computes quantities associated with the Gaussian linear state space model.

class `quantecon.lss.LinearStateSpace`(*A*, *C*, *G*, *H=None*, *mu_0=None*, *Sigma_0=None*)

Bases: `object`

A class that describes a Gaussian linear state space model of the form:

x_{t+1} = A x_t + C w_{t+1}

y_t = G x_t + H v_t

where {w_t} and {v_t} are independent and standard normal with dimensions k and l respectively. The initial conditions are mu_0 and Sigma_0 for x_0 ~ N(mu_0, Sigma_0). When Sigma_0=0, the draw of x_0 is exactly mu_0.

Parameters **A** : array_like or scalar(float)

Part of the state transition equation. It should be *n x n*

C : array_like or scalar(float)

Part of the state transition equation. It should be *n x m*

G : array_like or scalar(float)

Part of the observation equation. It should be *k x n*

H : array_like or scalar(float), optional(default=None)

Part of the observation equation. It should be *k x l*

mu_0 : array_like or scalar(float), optional(default=None)

This is the mean of initial draw and is *n x 1*

Sigma_0 : array_like or scalar(float), optional(default=None)

This is the variance of the initial draw and is *n x n* and also should be positive definite and symmetric

Attributes

A, C, G, H, mu_0, Sigma_0	(see Parameters)
n, k, m, l	(scalar(int)) The dimensions of x_t, y_t, w_t and v_t respectively

Methods

`convert`(x)	Convert array_like objects (lists of lists, floats, etc.) into
`geometric_sums`(beta, x_t)	Forecast the geometric sums
`impulse_response`([j])	Pulls off the imuplse response coefficients to a shock
`moment_sequence`()	Create a generator to calculate the population mean and variance-convariance matrix for both x_t and y_t, starting at the initial condition (self.mu_0, self.Sigma_0).
`replicate`([T, num_reps])	Simulate num_reps observations of x_T and y_T given x_0 ~ N(mu_0, Sigma_0).
`simulate`([ts_length])	Simulate a time series of length ts_length, first drawing
`stationary_distributions`([max_iter, tol])	Compute the moments of the stationary distributions of x_t and y_t if possible.

convert (*x*)

Convert array_like objects (lists of lists, floats, etc.) into well formed 2D NumPy arrays

geometric_sums (*beta*, *x_t*)

Forecast the geometric sums

S_x := E [sum_{j=0}^{infty} beta^j x_{t+j} | x_t]

S_y := E [sum_{j=0}^{infty} beta^j y_{t+j} | x_t]

Parameters **beta** : scalar(float)

Discount factor, in [0, 1)

beta : array_like(float)

The term x_t for conditioning

Returns **S_x** : array_like(float)

Geometric sum as defined above

S_y : array_like(float)

Geometric sum as defined above

impulse_response (*j=5*)

Pulls off the imuplse response coefficients to a shock in w_{t} for x and y

Important to note: We are uninterested in the shocks to v for this method

- x coefficients are C, AC, A^2 C...
- y coefficients are GC, GAC, GA^2C...

Parameters **j** : Scalar(int)

Number of coefficients that we want

Returns **xcoef** : list(array_like(float, 2))

The coefficients for x

ycoef : list(array_like(float, 2))

The coefficients for y

moment_sequence()

Create a generator to calculate the population mean and variance-convariance matrix for both x_t and y_t, starting at the initial condition (self.mu_0, self.Sigma_0). Each iteration produces a 4-tuple of items (mu_x, mu_y, Sigma_x, Sigma_y) for the next period.

Yields **mu_x** : array_like(float)

An n x 1 array representing the population mean of x_t

mu_y : array_like(float)

A k x 1 array representing the population mean of y_t

Sigma_x : array_like(float)

An n x n array representing the variance-covariance matrix of x_t

Sigma_y : array_like(float)

A k x k array representing the variance-covariance matrix of y_t

replicate(*T=10*, *num_reps=100*)

Simulate num_reps observations of x_T and y_T given x_0 ~ N(mu_0, Sigma_0).

Parameters **T** : scalar(int), optional(default=10)

The period that we want to replicate values for

num_reps : scalar(int), optional(default=100)

The number of replications that we want

Returns **x** : array_like(float)

An n x num_reps array, where the j-th column is the j_th observation of x_T

y : array_like(float)

A k x num_reps array, where the j-th column is the j_th observation of y_T

simulate(*ts_length=100*)

Simulate a time series of length ts_length, first drawing

x_0 ~ N(mu_0, Sigma_0)

Parameters **ts_length** : scalar(int), optional(default=100)

The length of the simulation

Returns **x** : array_like(float)

An n x ts_length array, where the t-th column is x_t

y : array_like(float)

A k x ts_length array, where the t-th column is y_t

stationary_distributions(*max_iter=200*, *tol=1e-05*)

Compute the moments of the stationary distributions of x_t and y_t if possible. Computation is by iteration, starting from the initial conditions self.mu_0 and self.Sigma_0

Parameters **max_iter** : scalar(int), optional(default=200)

The maximum number of iterations allowed

tol : scalar(float), optional(default=1e-5)

The tolerance level that one wishes to achieve

Returns **mu_x_star** : array_like(float)

An n x 1 array representing the stationary mean of x_t

mu_y_star : array_like(float)

An k x 1 array representing the stationary mean of y_t

Sigma_x_star : array_like(float)

An n x n array representing the stationary var-cov matrix of x_t

Sigma_y_star : array_like(float)

An k x k array representing the stationary var-cov matrix of y_t

`quantecon.lss.`**`multivariate_normal`**(*mean*, *cov*[, *size*])

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

Parameters **mean** : 1-D array_like, of length N

Mean of the N-dimensional distribution.

cov : 2-D array_like, of shape (N, N)

Covariance matrix of the distribution. It must be symmetric and positive-semidefinite for proper sampling.

size : int or tuple of ints, optional

Given a shape of, for example, `(m,n,k)`, `m*n*k` samples are generated, and packed in an *m*-by-*n*-by-*k* arrangement. Because each sample is *N*-dimensional, the output shape is `(m,n,k,N)`. If no shape is specified, a single (*N*-D) sample is returned.

Returns **out** : ndarray

The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

Notes

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ... x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j. The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- Spherical covariance (*cov* is a multiple of the identity matrix)
- Diagonal covariance (*cov* has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0, 0]
>>> cov = [[1, 0], [0, 100]]  # diagonal covariance
```

Diagonal covariance means that points are oriented along x or y-axis:

```
>>> import matplotlib.pyplot as plt
>>> x, y = np.random.multivariate_normal(mean, cov, 5000).T
>>> plt.plot(x, y, 'x')
>>> plt.axis('equal')
>>> plt.show()
```

Note that the covariance matrix must be positive semidefinite (a.k.a. nonnegative-definite). Otherwise, the behavior of this method is undefined and backwards compatibility is not guaranteed.

References

[R11], *[R12]*

Examples

```
>>> mean = (1, 2)
>>> cov = [[1, 0], [0, 1]]
>>> x = np.random.multivariate_normal(mean, cov, (3, 3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> list((x[0,0,:] - mean) < 0.6)
[True, True]
```

`quantecon.lss.`**`simulate_linear_model`**

This is a separate function for simulating a vector linear system of the form

x_{t+1} = A x_t + v_t given x_0 = x0

Here x_t and v_t are both n x 1 and A is n x n.

The purpose of separating this functionality out is to target it for optimization by Numba. For the same reason, matrix multiplication is broken down into for loops.

Parameters **A** : array_like or scalar(float)

Should be n x n

x0 : array_like

Should be n x 1. Initial condition

v : np.ndarray

Should be n x ts_length-1. Its t-th column is used as the time t shock v_t

ts_length : int

The length of the time series

Returns **x** : np.ndarray

Time series with ts_length columns, the t-th column being x_t

4.17 matrix_eqn

Filename: matrix_eqn.py

This files holds several functions that are used to solve matrix equations. Currently has functionality to solve:

- Lyapunov Equations
- Ricatti Equations

TODO: 1. See issue 47 on github repository, should add support for Sylvester equations 2. Fix warnings from checking conditioning of matrices

`quantecon.matrix_eqn.`**`solve_discrete_lyapunov`**(*A*, *B*, *max_it=50*, *method='doubling'*)

Computes the solution to the discrete lyapunov equation

$$AXA' - X + B = 0$$

X is computed by using a doubling algorithm. In particular, we iterate to convergence on X_j with the following recursions for j = 1, 2,... starting from X_0 = B, a_0 = A:

$$a_j = a_{j-1} a_{j-1}$$
$$X_j = X_{j-1} + a_{j-1} X_{j-1} a_{j-1}'$$

Parameters A : array_like(float, ndim=2)

An n x n matrix as described above. We assume in order for convergence that the eigenvalues of A have moduli bounded by unity

B : array_like(float, ndim=2)

An n x n matrix as described above. We assume in order for convergence that the eigenvalues of A have moduli bounded by unity

max_it : scalar(int), optional(default=50)

The maximum number of iterations

method : string, optional(default="doubling")

Describes the solution method to use. If it is "doubling" then uses the doubling algorithm to solve, if it is "bartels-stewart" then it uses scipy's implementation of the Bartels-Stewart approach.

Returns gamma1: array_like(float, ndim=2)

Represents the value V

`quantecon.matrix_eqn.`**`solve_discrete_riccati`**(*A*, *B*, *Q*, *R*, *N=None*, *tolerance=1e-10*, *max_iter=500*)

Solves the discrete-time algebraic Riccati equation

X = A'XA - (N + B'XA)'(B'XB + R)^{-1}(N + B'XA) + Q

via a modified structured doubling algorithm. An explanation of the algorithm can be found in the reference below.

Note that SciPy also has a discrete riccati equation solver. However it cannot handle the case where R is not invertible, or when N is nonzero. Both of these cases can be handled in the algorithm implemented below.

Parameters A : array_like(float, ndim=2) k x k array.

B : array_like(float, ndim=2)

k x n array

Q : array_like(float, ndim=2)

k x k, should be symmetric and non-negative definite

R : array_like(float, ndim=2)

n x n, should be symmetric and positive definite

N : array_like(float, ndim=2)

n x k array

tolerance : scalar(float), optional(default=1e-10)

The tolerance level for convergence

max_iter : scalar(int), optional(default=500)

The maximum number of iterations allowed

Returns X : array_like(float, ndim=2)

The fixed point of the Riccati equation; a k x k array representing the approximate solution

References

Chiang, Chun-Yueh, Hung-Yuan Fan, and Wen-Wei Lin. "STRUCTURED DOUBLING ALGORITHM FOR DISCRETE-TIME ALGEBRAIC RICCATI EQUATIONS WITH SINGULAR CONTROL WEIGHTING MATRICES." Taiwanese Journal of Mathematics 14, no. 3A (2010): pp-935.

4.18 quad

Filename: quad.py Authors: Chase Coleman, Spencer Lyon Date: 2014-07-01

Defining various quadrature routines.

Based on the quadrature routines found in the CompEcon toolbox by Miranda and Fackler.

4.18.1 References

Miranda, Mario J, and Paul L Fackler. Applied Computational Economics and Finance, MIT Press, 2002.

`quantecon.quad.`**`qnwcheb`**(*n*, *a=1*, *b=1*)

Computes multivariate Guass-Checbychev quadrature nodes and weights.

Parameters n : int or array_like(float)

A length-d iterable of the number of nodes in each dimension

a : scalar or array_like(float)

A length-d iterable of lower endpoints. If a scalar is given, that constant is repeated d times, where d is the number of dimensions

b : scalar or array_like(float)

A length-d iterable of upper endpoints. If a scalar is given, that constant is repeated d times, where d is the number of dimensions

Returns nodes : np.ndarray(dtype=float)

Quadrature nodes

weights : np.ndarray(dtype=float)

Weights for quadrature nodes

Notes

Based of original function `qnwcheb` in CompEcon toolbox by Miranda and Fackler

References

Miranda, Mario J, and Paul L Fackler. Applied Computational Economics and Finance, MIT Press, 2002.

`quantecon.quad.`**`qnwequi`**(*n*, *a*, *b*, *kind='N'*, *equidist_pp=None*)

Generates equidistributed sequences with property that averages value of integrable function evaluated over the sequence converges to the integral as n goes to infinity.

Parameters n : int

Number of sequence points

a : scalar or array_like(float)

A length-d iterable of lower endpoints. If a scalar is given, that constant is repeated d times, where d is the number of dimensions

b : scalar or array_like(float)

A length-d iterable of upper endpoints. If a scalar is given, that constant is repeated d times, where d is the number of dimensions

kind : string, optional(default="N")

One of the following:

- N - Neiderreiter (default)
- W - Weyl
- H - Haber
- R - pseudo Random

equidist_pp : array_like, optional(default=None)

TODO: I don't know what this does

Returns nodes : np.ndarray(dtype=float)

Quadrature nodes

weights : np.ndarray(dtype=float)

Weights for quadrature nodes

Notes

Based of original function `qnwequi` in CompEcon toolbox by Miranda and Fackler

References

Miranda, Mario J, and Paul L Fackler. Applied Computational Economics and Finance, MIT Press, 2002.

`quantecon.quad.`**`qnwlege`**(*n*, *a*, *b*)

Computes multivariate Guass-Legendre quadrature nodes and weights.

Parameters **n** : int or array_like(float)

A length-d iterable of the number of nodes in each dimension

a : scalar or array_like(float)

A length-d iterable of lower endpoints. If a scalar is given, that constant is repeated d times, where d is the number of dimensions

b : scalar or array_like(float)

A length-d iterable of upper endpoints. If a scalar is given, that constant is repeated d times, where d is the number of dimensions

Returns **nodes** : np.ndarray(dtype=float)

Quadrature nodes

weights : np.ndarray(dtype=float)

Weights for quadrature nodes

Notes

Based of original function `qnwlege` in CompEcon toolbox by Miranda and Fackler

References

Miranda, Mario J, and Paul L Fackler. Applied Computational Economics and Finance, MIT Press, 2002.

`quantecon.quad.`**`qnwnorm`**(*n*, *mu=None*, *sig2=None*, *usesqrtm=False*)

Computes nodes and weights for multivariate normal distribution

Parameters **n** : int or array_like(float)

A length-d iterable of the number of nodes in each dimension

mu : scalar or array_like(float), optional(default=zeros(d))

The means of each dimension of the random variable. If a scalar is given, that constant is repeated d times, where d is the number of dimensions

sig2 : array_like(float), optional(default=eye(d))

A d x d array representing the variance-covariance matrix of the multivariate normal distribution.

Returns **nodes** : np.ndarray(dtype=float)

Quadrature nodes

weights : np.ndarray(dtype=float)

Weights for quadrature nodes

Notes

Based of original function `qnwnorm` in CompEcon toolbox by Miranda and Fackler

References

Miranda, Mario J, and Paul L Fackler. Applied Computational Economics and Finance, MIT Press, 2002.

`quantecon.quad.`**`qnwlogn`**(*n*, *mu=None*, *sig2=None*)

Computes nodes and weights for multivariate lognormal distribution

Parameters n : int or array_like(float)

A length-d iterable of the number of nodes in each dimension

mu : scalar or array_like(float), optional(default=zeros(d))

The means of each dimension of the random variable. If a scalar is given, that constant is repeated d times, where d is the number of dimensions

sig2 : array_like(float), optional(default=eye(d))

A d x d array representing the variance-covariance matrix of the multivariate normal distribution.

Returns nodes : np.ndarray(dtype=float)

Quadrature nodes

weights : np.ndarray(dtype=float)

Weights for quadrature nodes

Notes

Based of original function `qnwlogn` in CompEcon toolbox by Miranda and Fackler

References

Miranda, Mario J, and Paul L Fackler. Applied Computational Economics and Finance, MIT Press, 2002.

`quantecon.quad.`**`qnwsimp`**(*n*, *a*, *b*)

Computes multivariate Simpson quadrature nodes and weights.

Parameters n : int or array_like(float)

A length-d iterable of the number of nodes in each dimension

a : scalar or array_like(float)

A length-d iterable of lower endpoints. If a scalar is given, that constant is repeated d times, where d is the number of dimensions

b : scalar or array_like(float)

A length-d iterable of upper endpoints. If a scalar is given, that constant is repeated d times, where d is the number of dimensions

Returns **nodes** : np.ndarray(dtype=float)

Quadrature nodes

weights : np.ndarray(dtype=float)

Weights for quadrature nodes

Notes

Based of original function `qnwsimp` in CompEcon toolbox by Miranda and Fackler

References

Miranda, Mario J, and Paul L Fackler. Applied Computational Economics and Finance, MIT Press, 2002.

`quantecon.quad.`**`qnwtrap`**`(`*n*, *a*, *b*`)`

Computes multivariate trapezoid rule quadrature nodes and weights.

Parameters **n** : int or array_like(float)

A length-d iterable of the number of nodes in each dimension

a : scalar or array_like(float)

A length-d iterable of lower endpoints. If a scalar is given, that constant is repeated d times, where d is the number of dimensions

b : scalar or array_like(float)

A length-d iterable of upper endpoints. If a scalar is given, that constant is repeated d times, where d is the number of dimensions

Returns **nodes** : np.ndarray(dtype=float)

Quadrature nodes

weights : np.ndarray(dtype=float)

Weights for quadrature nodes

Notes

Based of original function `qnwtrap` in CompEcon toolbox by Miranda and Fackler

References

Miranda, Mario J, and Paul L Fackler. Applied Computational Economics and Finance, MIT Press, 2002.

`quantecon.quad.`**`qnwunif`**`(`*n*, *a*, *b*`)`

Computes quadrature nodes and weights for multivariate uniform distribution

Parameters **n** : int or array_like(float)

A length-d iterable of the number of nodes in each dimension

a : scalar or array_like(float)

A length-d iterable of lower endpoints. If a scalar is given, that constant is repeated d times, where d is the number of dimensions

b : scalar or array_like(float)

A length-d iterable of upper endpoints. If a scalar is given, that constant is repeated d times, where d is the number of dimensions

Returns nodes : np.ndarray(dtype=float)

Quadrature nodes

weights : np.ndarray(dtype=float)

Weights for quadrature nodes

Notes

Based of original function `qnwunif` in CompEcon toolbox by Miranda and Fackler

References

Miranda, Mario J, and Paul L Fackler. Applied Computational Economics and Finance, MIT Press, 2002.

`quantecon.quad.`**`quadrect`** (*f*, *n*, *a*, *b*, *kind='lege'*, **args*, ***kwargs*)

Integrate the d-dimensional function f on a rectangle with lower and upper bound for dimension i defined by a[i] and b[i], respectively; using n[i] points.

Parameters f : function

The function to integrate over. This should be a function that accepts as its first argument a matrix representing points along each dimension (each dimension is a column). Other arguments that need to be passed to the function are caught by **args* and ***kwargs*

n : int or array_like(float)

A length-d iterable of the number of nodes in each dimension

a : scalar or array_like(float)

A length-d iterable of lower endpoints. If a scalar is given, that constant is repeated d times, where d is the number of dimensions

b : scalar or array_like(float)

A length-d iterable of upper endpoints. If a scalar is given, that constant is repeated d times, where d is the number of dimensions

kind : string, optional(default='lege')

Specifies which type of integration to perform. Valid values are:

lege - Gauss-Legendre cheb - Gauss-Chebyshev trap - trapezoid rule simp - Simpson rule N - Neiderreiter equidistributed sequence W - Weyl equidistributed sequence H - Haber equidistributed sequence R - Monte Carlo

***args, **kwargs :**

Other arguments passed to the function f

Returns out : scalar (float)

The value of the integral on the region [a, b]

Notes

Based of original function `quadrect` in CompEcon toolbox by Miranda and Fackler

References

Miranda, Mario J, and Paul L Fackler. Applied Computational Economics and Finance, MIT Press, 2002.

`quantecon.quad.`**`qnwbeta`**(*n*, *a=1.0*, *b=1.0*)

Computes nodes and weights for beta distribution

Parameters **n** : int or array_like(float)

A length-d iterable of the number of nodes in each dimension

a : scalar or array_like(float), optional(default=1.0)

A length-d

b : array_like(float), optional(default=1.0)

A d x d array representing the variance-covariance matrix of the multivariate normal distribution.

Returns **nodes** : np.ndarray(dtype=float)

Quadrature nodes

weights : np.ndarray(dtype=float)

Weights for quadrature nodes

Notes

Based of original function `qnwbeta` in CompEcon toolbox by Miranda and Fackler

References

Miranda, Mario J, and Paul L Fackler. Applied Computational Economics and Finance, MIT Press, 2002.

`quantecon.quad.`**`qnwgamma`**(*n*, *a=None*)

Computes nodes and weights for gamma distribution

Parameters **n** : int or array_like(float)

A length-d iterable of the number of nodes in each dimension

mu : scalar or array_like(float), optional(default=zeros(d))

The means of each dimension of the random variable. If a scalar is given, that constant is repeated d times, where d is the number of dimensions

sig2 : array_like(float), optional(default=eye(d))

A d x d array representing the variance-covariance matrix of the multivariate normal distribution.

Returns **nodes** : np.ndarray(dtype=float)

Quadrature nodes

weights : np.ndarray(dtype=float)

Weights for quadrature nodes

Notes

Based of original function `qnwgamma` in CompEcon toolbox by Miranda and Fackler

References

Miranda, Mario J, and Paul L Fackler. Applied Computational Economics and Finance, MIT Press, 2002.

4.19 quadsums

Filename: quadsums.py

Authors: Thomas Sargent, John Stachurski

This module provides functions to compute quadratic sums of the form described in the docstrings.

`quantecon.quadsums.`**`m_quadratic_sum`**(*A*, *B*, *max_it=50*)

Computes the quadratic sum

$$V = \sum_{j=0}^{\infty} A^j B A^{j'}$$

V is computed by solving the corresponding discrete lyapunov equation using the doubling algorithm. See the documentation of *util.solve_discrete_lyapunov* for more information.

Parameters **A** : array_like(float, ndim=2)

An n x n matrix as described above. We assume in order for convergence that the eigenvalues of A have moduli bounded by unity

B : array_like(float, ndim=2)

An n x n matrix as described above. We assume in order for convergence that the eigenvalues of A have moduli bounded by unity

max_it : scalar(int), optional(default=50)

The maximum number of iterations

Returns gamma1: array_like(float, ndim=2)

Represents the value V

`quantecon.quadsums.`**`var_quadratic_sum`**(*A*, *C*, *H*, *beta*, *x0*)

Computes the expected discounted quadratic sum

$$q(x_0) = E \sum_{t=0}^{\infty} \beta^t x_t' H x_t$$

Here {x_t} is the VAR process x_{t+1} = A x_t + C w_t with {w_t} standard normal and x_0 the initial condition.

Parameters **A** : array_like(float, ndim=2)

The matrix described above in description. Should be n x n

C : array_like(float, ndim=2)

The matrix described above in description. Should be n x n

H : array_like(float, ndim=2)

The matrix described above in description. Should be n x n

beta: scalar(float)

Should take a value in (0, 1)

x_0: array_like(float, ndim=1)

The initial condtion. A conformable array (of length n, or with n rows)

Returns q0: scalar(float)

Represents the value q(x_0)

Remarks: The formula for computing q(x_0) is q(x_0) = x_0' Q x_0 + v

where

Q is the solution to Q = H + beta A' Q A and v = trace(C' Q C) beta / (1 - beta)

4.20 rank_nullspace

`quantecon.rank_nullspace.`**`nullspace`**(*A*, *atol=1e-13*, *rtol=0*)

Compute an approximate basis for the nullspace of A.

The algorithm used by this function is based on the singular value decomposition of *A*.

Parameters **A** : array_like(float, ndim=1 or 2)

A should be at most 2-D. A 1-D array with length k will be treated as a 2-D with shape (1, k)

atol : scalar(float), optional(default=1e-13)

The absolute tolerance for a zero singular value. Singular values smaller than *atol* are considered to be zero.

rtol : scalar(float), optional(default=0)

The relative tolerance. Singular values less than rtol*smax are considered to be zero, where smax is the largest singular value.

Returns **ns** : array_like(float, ndim=2)

If *A* is an array with shape (m, k), then *ns* will be an array with shape (k, n), where n is the estimated dimension of the nullspace of *A*. The columns of *ns* are a basis for the nullspace; each element in numpy.dot(A, ns) will be approximately zero.

Note: If both *atol* and *rtol* are positive, the combined tolerance

is the maximum of the two; that is:

tol = max(atol, rtol * smax)

Note: Singular values smaller than *tol* are considered to be zero.

`quantecon.rank_nullspace.`**`rank_est`**(*A*, *atol=1e-13*, *rtol=0*)

Estimate the rank (i.e. the dimension of the nullspace) of a matrix.

The algorithm used by this function is based on the singular value decomposition of *A*.

Parameters **A** : array_like(float, ndim=1 or 2)

A should be at most 2-D. A 1-D array with length n will be treated as a 2-D with shape (1, n)

atol : scalar(float), optional(default=1e-13)

The absolute tolerance for a zero singular value. Singular values smaller than *atol* are considered to be zero.

rtol : scalar(float), optional(default=0)

The relative tolerance. Singular values less than rtol*smax are considered to be zero, where smax is the largest singular value.

Returns **r** : scalar(int)

The estimated rank of the matrix.

Note: If both *atol* and *rtol* are positive, the combined tolerance

is the maximum of the two; that is:

tol = max(atol, rtol * smax)

Note: Singular values smaller than *tol* are considered to be zero.

See also:

`numpy.linalg.matrix_rank` matrix_rank is basically the same as this function, but it does not provide the option of the absolute tolerance.

4.21 robustlq

Filename: robustlq.py

Authors: Chase Coleman, Spencer Lyon, Thomas Sargent, John Stachurski

Solves robust LQ control problems.

class `quantecon.robustlq.`**`RBLQ`**(*Q*, *R*, *A*, *B*, *C*, *beta*, *theta*)

Bases: `object`

Provides methods for analysing infinite horizon robust LQ control problems of the form

$$min_{u_t} sum_t beta^t x_t' R x_t + u_t' Q u_t$$

subject to

$$x_{t+1} = Ax_t + Bu_t + Cw_{t+1}$$

and with model misspecification parameter theta.

Parameters **Q** : array_like(float, ndim=2)

The cost(payoff) matrix for the controls. See above for more. Q should be k x k and symmetric and positive definite

R : array_like(float, ndim=2)

The cost(payoff) matrix for the state. See above for more. R should be n x n and symmetric and non-negative definite

A : array_like(float, ndim=2)

The matrix that corresponds with the state in the state space system. A should be n x n

B : array_like(float, ndim=2)

The matrix that corresponds with the control in the state space system. B should be n x k

C : array_like(float, ndim=2)

The matrix that corresponds with the random process in the state space system. C should be n x j

beta : scalar(float)

The discount factor in the robust control problem

theta : scalar(float)

The robustness factor in the robust control problem

Attributes

Q, R, A, B, C, beta, theta	(see Parameters)
k, n, j	(scalar(int)) The dimensions of the matrices

Methods

F_to_K(F)	Compute agent 2's best cost-minimizing response K, given F.
K_to_F(K)	Compute agent 1's best value-maximizing response F, given K.
b_operator(P)	The B operator, mapping P into
compute_deterministic_entropy(F, K, x0)	Given K and F, compute the value of deterministic entropy, which is sum_t beta^t x_t' K'K x_t with x_{t+1} = (A - BF + CK) x_t.
d_operator(P)	The D operator, mapping P into
evaluate_F(F)	Given a fixed policy F, with the interpretation u = -F x, this function computes the matrix P_F and constant d_F associated with discounted cost J_F(x) = x' P_F x + d_F.
robust_rule()	This method solves the robust control problem by tricking it
robust_rule_simple([P_init, max_iter, tol])	A simple algorithm for computing the robust policy F and the corresponding value function P, based around straightforward iteration with the robust Bellman operator.

F_to_K(*F*)

Compute agent 2's best cost-minimizing response K, given F.

Parameters **F** : array_like(float, ndim=2)

A k x n array

Returns **K** : array_like(float, ndim=2)

Agent's best cost minimizing response for a given F

P : array_like(float, ndim=2)

The value function for a given F

K_to_F(*K*)

Compute agent 1's best value-maximizing response F, given K.

Parameters **K** : array_like(float, ndim=2)

A j x n array

Returns **F** : array_like(float, ndim=2)

The policy function for a given K

P : array_like(float, ndim=2)

The value function for a given K

b_operator(*P*)

The B operator, mapping P into

$$B(P) := R - beta^2 A'PB(Q + betaB'PB)^{-1}B'PA + betaA'PA$$

and also returning

$$F := (Q + betaB'PB)^{-1}betaB'PA$$

Parameters **P** : array_like(float, ndim=2)

A matrix that should be n x n

Returns **F** : array_like(float, ndim=2)

The F matrix as defined above

new_p : array_like(float, ndim=2)

The matrix P after applying the B operator

compute_deterministic_entropy(*F*, *K*, *x0*)

Given K and F, compute the value of deterministic entropy, which is sum_t beta^t x_t' K'K x_t with x_{t+1} = (A - BF + CK) x_t.

Parameters **F** : array_like(float, ndim=2)

The policy function, a k x n array

K : array_like(float, ndim=2)

The worst case matrix, a j x n array

x0 : array_like(float, ndim=1)

The initial condition for state

Returns **e** : scalar(int)

The deterministic entropy

d_operator(*P*)

The D operator, mapping P into

$$D(P) := P + PC(thetaI - C'PC)^{-1}C'P.$$

Parameters **P** : array_like(float, ndim=2)

A matrix that should be n x n

Returns **dP** : array_like(float, ndim=2)

The matrix P after applying the D operator

evaluate_F(*F*)

Given a fixed policy F, with the interpretation u = -F x, this function computes the matrix P_F and constant d_F associated with discounted cost J_F(x) = x' P_F x + d_F.

Parameters **F** : array_like(float, ndim=2)

The policy function, a k x n array

Returns **P_F** : array_like(float, ndim=2)

Matrix for discounted cost

d_F : scalar(float)

Constant for discounted cost

K_F : array_like(float, ndim=2)

Worst case policy

O_F : array_like(float, ndim=2)

Matrix for discounted entropy

o_F : scalar(float)

Constant for discounted entropy

robust_rule()

This method solves the robust control problem by tricking it into a stacked LQ problem, as described in chapter 2 of Hansen- Sargent's text "Robustness." The optimal control with observed state is

$$u_t = -Fx_t$$

And the value function is -x'Px

Returns **F** : array_like(float, ndim=2)

The optimal control matrix from above

P : array_like(float, ndim=2)

The positive semi-definite matrix defining the value function

K : array_like(float, ndim=2)

the worst-case shock matrix K, where $w_{t+1} = Kx_t$ is the worst case shock

robust_rule_simple(*P_init=None*, *max_iter=80*, *tol=1e-08*)

A simple algorithm for computing the robust policy F and the corresponding value function P, based around straightforward iteration with the robust Bellman operator. This function is easier to understand but one or two orders of magnitude slower than self.robust_rule(). For more information see the docstring of that method.

Parameters **P_init** : array_like(float, ndim=2), optional(default=None)

The initial guess for the value function matrix. It will be a matrix of zeros if no guess is given

max_iter : scalar(int), optional(default=80)

The maximum number of iterations that are allowed

tol : scalar(float), optional(default=1e-8)

The tolerance for convergence

Returns **F** : array_like(float, ndim=2)

The optimal control matrix from above

P : array_like(float, ndim=2)

The positive semi-definite matrix defining the value function

K : array_like(float, ndim=2)

the worst-case shock matrix K, where $w_{t+1} = Kx_t$ is the worst case shock

CHAPTER 5

Utilities

5.1 array

5.1.1 Array Utilities

Array

searchsorted

`quantecon.util.array.`**`searchsorted`**

Custom version of np.searchsorted. Return the largest index i such that $a[i-1] <= v < a[i]$ (for $i = 0$, $v < a[0]$); if $v[n-1] <= v$, return n, where $n = len(a)$.

Parameters **a** : ndarray(float, ndim=1)

Input array. Must be sorted in ascending order.

v : scalar(float)

Value to be compared with elements of *a*.

Returns scalar(int)

Largest index i such that $a[i-1] <= v < a[i]$, or len(a) if no such index exists.

Notes

This routine is jit-complied if the module Numba is vailable; if not, it is an alias of np.searchsorted(a, v, side='right').

Examples

```
>>> a = np.array([0.2, 0.4, 1.0])
>>> searchsorted(a, 0.1)
0
>>> searchsorted(a, 0.4)
2
>>> searchsorted(a, 2)
3
```

5.2 common_messages

5.2.1 Warnings Module

Contains a collection of warning messages for consistent package wide notifications

5.3 notebooks

Support functions to Support QuantEcon.notebooks

The purpose of these utilities is to implement simple support functions to allow for automatic downloading of any support files (python modules, or data) that may be required to run demonstration notebooks.

5.3.1 Note

Files on the REMOTE Github Server can be organised into folders but they will end up at the root level of when downloaded as a support File

"https://github.com/QuantEcon/QuantEcon.notebooks/raw/master/dependencies/mpi/something.py" –> ./something.py

5.3.2 TODO

1. Write Style guide for QuantEcon.notebook contributions
2. Write an interface for Dat Server
3. Platform Agnostic (replace wget usage)

`quantecon.util.notebooks.`**`fetch_nb_dependencies`**(*files*, *repo='https://github.com/QuantEcon/QuantEcon.notebooks*, *raw='raw'*, *branch='master'*, *folder='dependencies'*, *overwrite=False*, *verbose=True*)

Retrieve raw files from QuantEcon.notebooks or any other Github repo

Parameters file_list list or dict

A list of files to specify a collection of filenames A dict of dir : list(files) to specify a directory

repo str, optional(default=REPO)

raw str, optional(defualt=RAW)

This is here in case github changes access to their raw files through web links

branch str, optional(default=BRANCH)

folder str, optional(default=FOLDER)

overwrite bool, optional(default=False)

verbose bool, optional(default=True)

5.4 numba

Utilities to support Numba jitted functions

5.5 random

Utilities to Support Random State Infrastructure

`quantecon.util.random.check_random_state(seed)`

Check the random state of a given seed.

If seed is None, return the RandomState singleton used by np.random. If seed is an int, return a new RandomState instance seeded with seed. If seed is already a RandomState instance, return it.

Otherwise raise ValueError.

5.6 timing

Filename: timing.py Authors: Pablo Winant Date: 10/16/14 Provides Matlab-like tic, tac and toc functions.

`quantecon.util.timing.tac()`

Prints and returns elapsed time since last tic, tac or toc.

`quantecon.util.timing.tic()`

Saves time for future use with tac or toc.

`quantecon.util.timing.toc()`

Prints and returns elapsed time since last tic, tac or toc.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Bibliography

[R1] B. Codenotti, S. De Rossi, and M. Pagan, "An Experimental Analysis of Lemke-Howson Algorithm," arXiv:0811.3247, 2008.

[R2] C. E. Lemke and J. T. Howson, "Equilibrium Points of Bimatrix Games," Journal of the Society for Industrial and Applied Mathematics (1964), 413-423.

[R3] B. von Stengel, "Equilibrium Computation for Two-Player Games in Strategic and Extensive Form," Chapter 3, N. Nisan, T. Roughgarden, E. Tardos, and V. Vazirani eds., Algorithmic Game Theory, 2007.

[R4] A. McLennan and R. Tourky, "From Imitation Games to Kakutani," 2006.

[R5] Y. Rinott and M. Scarsini, "On the Number of Pure Strategy Nash Equilibria in Random Games," Games and Economic Behavior (2000), 274-293.

[R6] W. K. Grassmann, M. I. Taksar and D. P. Heyman, "Regenerative Analysis and Steady State Distributions for Markov Chains," Operations Research (1985), 1107-1116.

[R7] W. J. Stewart, Probability, Markov Chains, Queues, and Simulation, Princeton University Press, 2009.

[R8] A. McLennan and R. Tourky, "From Imitation Games to Kakutani," 2006.

[R9] Strongly connected component, Wikipedia.

[R10] Aperiodic graph, Wikipedia.

[R11] Papoulis, A., "Probability, Random Variables, and Stochastic Processes," 3rd ed., New York: McGraw-Hill, 1991.

[R12] Duda, R. O., Hart, P. E., and Stork, D. G., "Pattern Classification," 2nd ed., New York: Wiley, 2001.

Python Module Index

q

Index

G

I

K

L

M

N

O

P

Q

R

S

T

U

V

W