

---

# **QuantEcon Documentation**

*Release 0.3.8*

**QuantEcon Developer Team**

**Jun 06, 2018**



---

# Contents

---

<b>1</b>	<b>Game theory</b>	<b>3</b>
1.1	lemke_howson . . . . .	3
1.2	mclennan_tourky . . . . .	4
1.3	normal_form_game . . . . .	6
1.4	pure_nash . . . . .	12
1.5	random . . . . .	12
1.6	support_enumeration . . . . .	13
1.7	utilities . . . . .	14
1.8	vertex_enumeration . . . . .	15
1.9	bimatrix_generators . . . . .	15
<b>2</b>	<b>Markov</b>	<b>21</b>
2.1	approximation . . . . .	21
2.2	core . . . . .	22
2.3	ddp . . . . .	26
2.4	gth_solve . . . . .	34
2.5	random . . . . .	34
2.6	utilities . . . . .	36
<b>3</b>	<b>Random</b>	<b>37</b>
3.1	utilities . . . . .	37
<b>4</b>	<b>Tools</b>	<b>39</b>
4.1	arma . . . . .	39
4.2	ce_util . . . . .	41
4.3	compute_fp . . . . .	42
4.4	discrete_rv . . . . .	43
4.5	distributions . . . . .	45
4.6	ecdf . . . . .	46
4.7	estspec . . . . .	46
4.8	graph_tools . . . . .	47
4.9	gridtools . . . . .	50
4.10	ivp . . . . .	52
4.11	kalman . . . . .	54
4.12	lae . . . . .	57
4.13	lqcontrol . . . . .	57
4.14	lqnash . . . . .	60

4.15	lss . . . . .	61
4.16	matrix_eqn . . . . .	65
4.17	quad . . . . .	67
4.18	quadsums . . . . .	73
4.19	rank_nullspace . . . . .	74
4.20	robustlq . . . . .	75
<b>5</b>	<b>Utilities</b>	<b>79</b>
5.1	array . . . . .	79
5.2	combinatorics . . . . .	80
5.3	common_messages . . . . .	81
5.4	notebooks . . . . .	81
5.5	numba . . . . .	82
5.6	random . . . . .	83
5.7	timing . . . . .	83
<b>6</b>	<b>Indices and tables</b>	<b>85</b>
	<b>Bibliography</b>	<b>87</b>
	<b>Python Module Index</b>	<b>89</b>

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.



## 1.1 lemke\_howson

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 [2], implemented with “complementary pivoting” (see, e.g., von Stengel [3] for details).

### Parameters

**g** [NormalFormGame] NormalFormGame instance with 2 players.

**init\_pivot** [scalar(int), optional(default=0)] Initial pivot, an integer  $k$  such that  $0 \leq k < m+n$ , where integers  $0, \dots, m-1$  and  $m, \dots, m+n-1$  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. [1]; 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 [1]:
  - For  $k = \text{init\_pivot}, \text{init\_pivot} + 1, \dots, \text{init\_pivot} + (m+n-2)$ , (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  $\text{init\_pivot} + (m+n-1)$  (modulo  $m+n$ ) as the initial pivot, with a limit *max\_iter* on the total number of pivoting steps.

According 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 [1] suggests that *capping* be set to 10.

## References

[1], [2], [3]

## Examples

Consider the following game from von Stengel [3]:

```
>>> 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

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 [1].

### 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 integer (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

[1]

### 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

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*  $u_i: A_i \times A_{i+1} \times \dots \times A_{i+N-1} \rightarrow \mathbb{R}$  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  $A_{-i} = A_{i+1} \times \dots \times A_{i+N-1}$  and  $X_{-i} = X_{i+1} \times \dots \times X_{i+N-1}$ .

The (pure-action) *best response correspondence*  $b_i: X_{-i} \rightarrow A_i$  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  $x^* \in X_0 \times \dots \times X_{N-1}$  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:
```

(continues on next page)

(continued from previous page)

```
[[[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.

**payoff\_arrays** [tuple(ndarray(float, ndim=N))] Tuple of the payoff arrays, one for each player.

### Methods

---

<code>is_nash(action_profile[, tol])</code>	Return True if <i>action_profile</i> 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**

<i>best_response</i> (opponents_actions[, ...])	Return the best response action(s) to <i>opponents_actions</i> .
<i>dominated_actions</i> ([tol, method])	Return a list of actions that are strictly dominated by some mixed actions.
<i>is_best_response</i> (own_action, opponents_actions)	Return True if <i>own_action</i> is a best response to <i>opponents_actions</i> .
<i>is_dominated</i> (action[, tol, method])	Determine whether <i>action</i> is strictly dominated by some mixed action.
<i>payoff_vector</i> (opponents_actions)	Return an array of payoff values, one for each own action, given a profile of the opponents’ actions.
<i>random_choice</i> ([actions, random_state])	Return a pure action chosen randomly from <i>actions</i> .

**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 genera-

tor 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.

**dominated\_actions** (*tol=None, method=None*)

Return a list of actions that are strictly dominated by some mixed actions.

### Parameters

**tol** [scalar(float), optional(default=None)] Tolerance level used in determining domination. If None, default to the value of the *tol* attribute.

**method** [str, optional(default=None)] If None, *lemke\_howson* from *quantecon.game\_theory* is used to solve for a Nash equilibrium of an auxiliary zero-sum game. If *method* is set to *'simplex'* or *'interior-point'*, *scipy.optimize.linprog* is used with the method as specified by *method*.

### Returns

**list(int)** List of integers representing pure actions, each of which is strictly dominated by some mixed action.

**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.

**is\_dominated** (*action, tol=None, method=None*)

Determine whether *action* is strictly dominated by some mixed action.

### Parameters

**action** [scalar(int)] Integer representing a pure action.

**tol** [scalar(float), optional(default=None)] Tolerance level used in determining domination. If None, default to the value of the *tol* attribute.

**method** [str, optional(default=None)] If None, *lemke\_howson* from *quantecon.game\_theory* is used to solve for a Nash equilibrium of an auxiliary zero-sum game. If *method* is set to *'simplex'* or *'interior-point'*, *scipy.optimize.linprog* is used with the method as specified by *method*.

### Returns

**bool** True if *action* is strictly dominated by some mixed action; 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

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, tol=None)`

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

### Parameters

**g** [NormalFormGame]

**tol** [scalar(float), optional(default=None)] Tolerance level used in determining best responses. If None, default to the value of the *tol* attribute of *g*.

### 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 equilibrium:

```
>>> 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, tol=None)`

Generator version of *pure\_nash\_brute*.

### Parameters

**g** [NormalFormGame]

**tol** [scalar(float), optional(default=None)] Tolerance level used in determining best responses. If None, default to the value of the *tol* attribute of *g*.

### Yields

**out** [tuple(int)] Tuple of Nash equilibrium pure actions.

## 1.5 random

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 [1].



**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**

[1]

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

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 `scipy.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**

<code>clear()</code>	
<code>copy()</code>	
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get(k[,d])</code>	
<code>items()</code>	
<code>keys()</code>	
<code>pop(k[,d])</code>	If key is not found, d is returned if given, otherwise <code>KeyError</code> is raised
<code>popitem()</code>	2-tuple; but raise <code>KeyError</code> if D is empty.
<code>setdefault(k[,d])</code>	

Continued on next page

Table 3 – continued from previous page

<code>update([E, ]**F)</code>	If E is present and has a <code>.keys()</code> method, then does: for k in E: D[k] = E[k] If E is present and lacks a <code>.keys()</code> method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
<code>values()</code>	

## 1.8 vertex\_enumeration

Compute all mixed Nash equilibria of a 2-player normal form game by vertex enumeration.

### 1.8.1 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.vertex_enumeration.vertex_enumeration(g)`

Compute mixed-action Nash equilibria of a 2-player normal form game by enumeration and matching of vertices of the best response polytopes. For a non-degenerate game input, these are all the Nash equilibria.

Internally, `scipy.spatial.ConvexHull` is used to compute vertex enumeration of the best response polytopes, or equivalently, facet enumeration of their polar polytopes. Then, for each vertex of the polytope for player 0, vertices of the polytope for player 1 are searched to find a completely labeled pair.

#### 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.vertex_enumeration.vertex_enumeration_gen(g)`

Generator version of `vertex_enumeration`.

#### Parameters

`g` [NormalFormGame] NormalFormGame instance with 2 players.

#### Yields

`tuple(ndarray(float, ndim=1))` Tuple of Nash equilibrium mixed actions.

## 1.9 bimatrix\_generators

This module contains functions that generate NormalFormGame instances of the 2-player games studied by Fearnley, Igwe, and Savani (2015):

- Colonel Blotto Games (*blotto\_game*): A non-zero sum extension of the Blotto game as studied by Hortala-Vallve and Llorente-Saguer (2012), where opposing parties have asymmetric and heterogeneous battlefield valuations.
- Ranking Games (*ranking\_game*): In these games, as studied by Goldberg et al. (2013), each player chooses an effort level associated with a cost and a score. The players are ranked according to their scores, and the player with the higher score wins the prize. Each player’s payoff is given by the value of the prize minus the cost of the effort.

- SGC Games (*sgc\_game*): These games were introduced by Sandholm, Gilpin, and Conitzer (2005) as a worst case scenario for support enumeration as it has a unique equilibrium where each player uses half of his actions in his support.
- Tournament Games (*tournament\_game*): These games are constructed by Anbalagan et al. (2013) as games that do not have interim epsilon-Nash equilibria with constant cardinality supports for epsilon smaller than a certain threshold.
- Unit Vector Games (*unit\_vector\_game*): These games are games where the payoff matrix of one player consists of unit (column) vectors, used by Savani and von Stengel (2016) to construct instances that are hard, in terms of computational complexity, both for the Lemke-Howson and support enumeration algorithms.

Large part of the code here is based on the C code available at <https://github.com/bimatrix-games/bimatrix-generators> distributed under BSD 3-Clause License.

### 1.9.1 References

- Y. Anbalagan, S. Norin, R. Savani, and A. Vetta, “Polylogarithmic Supports Are Required for Approximate Well-Supported Nash Equilibria below 2/3,” WINE, 2013.
- J. Fearnley, T. P. Igwe, and R. Savani, “An Empirical Study of Finding Approximate Equilibria in Bimatrix Games,” International Symposium on Experimental Algorithms (SEA), 2015.
- L. A. Goldberg, P. W. Goldberg, P. Krysta, and C. Ventre, “Ranking Games that have Competitiveness-based Strategies”, Theoretical Computer Science, 2013.
- R. Hortala-Vallve and A. Llorente-Saguer, “Pure Strategy Nash Equilibria in Non-Zero Sum Colonel Blotto Games”, International Journal of Game Theory, 2012.
- T. Sandholm, A. Gilpin, and V. Conitzer, “Mixed-Integer Programming Methods for Finding Nash Equilibria,” AAAI, 2005.
- R. Savani and B. von Stengel, “Unit Vector Games,” International Journal of Economic Theory, 2016.

```
quantecon.game_theory.game_generators.bimatrix_generators.blotto_game(h, t,
                                                                    rho,
                                                                    mu=0,
                                                                    ran-
                                                                    dom_state=None)
```

Return a NormalFormGame instance of a 2-player non-zero sum Colonel Blotto game (Hortala-Vallve and Llorente-Saguer, 2012), where the players have an equal number  $t$  of troops to assign to  $h$  hills (so that the number of actions for each player is equal to  $(t+h-1) \text{ choose } (h-1) = (t+h-1)!/(t!(h-1)!)$ ). Each player has a value for each hill that he receives if he assigns strictly more troops to the hill than his opponent (ties are broken uniformly at random), where the values are drawn from a multivariate normal distribution with covariance  $\rho$ . Each player’s payoff is the sum of the values of the hills won by that player.

#### Parameters

- h** [scalar(int)] Number of hills.
- t** [scalar(int)] Number of troops.
- rho** [scalar(float)] Covariance of the players’ values of each hill. Must be in  $[-1, 1]$ .
- mu** [scalar(float), optional(default=0)] Mean of the players’ values of each hill.
- 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]

## Examples

```
>>> g = blotto_game(2, 3, 0.5, random_state=1234)
>>> g.players[0]
Player([[ -0.44861083, -1.08443468, -1.08443468, -1.08443468],
        [ 0.18721302, -0.44861083, -1.08443468, -1.08443468],
        [ 0.18721302,  0.18721302, -0.44861083, -1.08443468],
        [ 0.18721302,  0.18721302,  0.18721302, -0.44861083]])
>>> g.players[1]
Player([[ -1.20042463, -1.39708658, -1.39708658, -1.39708658],
        [-1.00376268, -1.20042463, -1.39708658, -1.39708658],
        [-1.00376268, -1.00376268, -1.20042463, -1.39708658],
        [-1.00376268, -1.00376268, -1.00376268, -1.20042463]])
```

quantecon.game\_theory.game\_generators.bimatrix\_generators.**ranking\_game**(*n*,  
*steps=10*,  
*ran-*  
*dom\_state=None*)

Return a NormalFormGame instance of (the 2-player version of) the “ranking game” studied by Goldberg et al. (2013), where each player chooses an effort level associated with a score and a cost which are both increasing functions with randomly generated step sizes. The player with the higher score wins the first prize, whose value is 1, and the other player obtains the “second prize” of value 0; in the case of a tie, the first prize is split and each player receives a value of 0.5. The payoff of a player is given by the value of the prize minus the cost of the effort.

### Parameters

**n** [scalar(int)] Number of actions, i.e. number of possible effort levels.

**steps** [scalar(int), optional(default=10)] Parameter determining the upper bound for the size of the random steps for the scores and costs for each player: The step sizes for the scores are drawn from  $1, \dots, steps$ , while those for the costs are multiples of  $1/(n*steps)$ , where the cost of effort level 0 is 0, and the maximum possible cost of effort level  $n-1$  is less than or equal to 1.

**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]

## Examples

```
>>> g = ranking_game(5, random_state=1234)
>>> g.players[0]
Player([[ 0. ,  0. ,  0. ,  0. ,  0. ],
        [ 0.82, -0.18, -0.18, -0.18, -0.18],
        [ 0.8 ,  0.8 , -0.2 , -0.2 , -0.2 ],
        [ 0.68,  0.68,  0.68, -0.32, -0.32],
        [ 0.66,  0.66,  0.66,  0.66, -0.34]])
>>> g.players[1]
```

(continues on next page)

(continued from previous page)

```
Player([[ 1. , 0. , 0. , 0. , 0. ],
        [ 0.8 , 0.8 , -0.2 , -0.2 , -0.2 ],
        [ 0.66, 0.66, 0.66, -0.34, -0.34],
        [ 0.6 , 0.6 , 0.6 , 0.6 , -0.4 ],
        [ 0.58, 0.58, 0.58, 0.58, 0.58]])
```

`quantecon.game_theory.game_generators.bimatrix_generators.sgc_game(k)`

Return a `NormalFormGame` instance of the 2-player game introduced by Sandholm, Gilpin, and Conitzer (2005), which has a unique Nash equilibrium, where each player plays half of the actions with positive probabilities. Payoffs are normalized so that the minimum and the maximum payoffs are 0 and 1, respectively.

#### Parameters

**k** [scalar(int)] Positive integer determining the number of actions. The returned game will have  $4*k-1$  actions for each player.

#### Returns

**g** [`NormalFormGame`]

#### Examples

```
>>> g = sgc_game(2)
>>> g.players[0]
Player([[ 0.75, 0.5 , 1. , 0.5 , 0.5 , 0.5 , 0.5 ],
        [ 1. , 0.75, 0.5 , 0.5 , 0.5 , 0.5 , 0.5 ],
        [ 0.5 , 1. , 0.75, 0.5 , 0.5 , 0.5 , 0.5 ],
        [ 0. , 0. , 0. , 0.75, 0. , 0. , 0. ],
        [ 0. , 0. , 0. , 0. , 0.75, 0. , 0. ],
        [ 0. , 0. , 0. , 0. , 0. , 0.75, 0. ],
        [ 0. , 0. , 0. , 0. , 0. , 0. , 0.75]])
>>> g.players[1]
Player([[ 0.75, 0.5 , 1. , 0.5 , 0.5 , 0.5 , 0.5 ],
        [ 1. , 0.75, 0.5 , 0.5 , 0.5 , 0.5 , 0.5 ],
        [ 0.5 , 1. , 0.75, 0.5 , 0.5 , 0.5 , 0.5 ],
        [ 0. , 0. , 0. , 0. , 0.75, 0. , 0. ],
        [ 0. , 0. , 0. , 0.75, 0. , 0. , 0. ],
        [ 0. , 0. , 0. , 0. , 0. , 0. , 0.75],
        [ 0. , 0. , 0. , 0. , 0. , 0.75, 0. ]])
```

`quantecon.game_theory.game_generators.bimatrix_generators.tournament_game(n, k, random_state=None)`

Return a `NormalFormGame` instance of the 2-player win-lose game, whose payoffs are either 0 or 1, introduced by Anbalagan et al. (2013). Player 0 has  $n$  actions, which constitute the set of nodes  $\{0, \dots, n-1\}$ , while player 1 has  $n$  choose  $k$  actions, each corresponding to a subset of  $k$  elements of the set of  $n$  nodes. Given a randomly generated tournament graph on the  $n$  nodes, the payoff for player 0 is 1 if, in the tournament, the node chosen by player 0 dominates all the nodes in the  $k$ -subset chosen by player 1. The payoff for player 1 is 1 if player 1's  $k$ -subset contains player 0's chosen node.

#### Parameters

**n** [scalar(int)] Number of nodes in the tournament graph.

**k** [scalar(int)] Size of subsets of nodes in the tournament graph.

**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]

### Notes

The actions of player 1 are ordered according to the combinatorial number system [1], which is different from the order used in the original library in C.

### References

[1]

### Examples

```
>>> g = tournament_game(5, 2, random_state=1234)
>>> g.players[0]
Player([[ 0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.],
        [ 1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  1.,  0.,  1.,  0.,  1.,  0.,  0.,  0.,  0.]])
>>> g.players[1]
Player([[ 1.,  1.,  0.,  0.,  0.],
        [ 1.,  0.,  1.,  0.,  0.],
        [ 0.,  1.,  1.,  0.,  0.],
        [ 1.,  0.,  0.,  1.,  0.],
        [ 0.,  1.,  0.,  1.,  0.],
        [ 0.,  0.,  1.,  1.,  0.],
        [ 1.,  0.,  0.,  0.,  1.],
        [ 0.,  1.,  0.,  0.,  1.],
        [ 0.,  0.,  1.,  0.,  1.],
        [ 0.,  0.,  0.,  1.,  1.]])
```

quantecon.game\_theory.game\_generators.bimatrix\_generators.**unit\_vector\_game**(*n*,  
*avoid\_pure\_nash=False*  
*ran-*  
*dom\_state=None*)

Return a NormalFormGame instance of the 2-player game “unit vector game” (Savani and von Stengel, 2016). Payoffs for player 1 are chosen randomly from the [0, 1) range. For player 0, each column contains exactly one 1 payoff and the rest is 0.

### Parameters

**n** [scalar(int)] Number of actions.

**avoid\_pure\_nash** [bool, optional(default=False)] If True, player 0’s payoffs will be placed in order to avoid pure Nash equilibria. (If necessary, the payoffs for player 1 are redrawn so as not to have a dominant action.)

**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]

### Examples

```
>>> g = unit_vector_game(4, random_state=1234)
>>> g.players[0]
Player([[ 1.,  0.,  1.,  0.],
        [ 0.,  0.,  0.,  1.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  1.,  0.,  0.]])
>>> g.players[1]
Player([[ 0.19151945,  0.62210877,  0.43772774,  0.78535858],
        [ 0.77997581,  0.27259261,  0.27646426,  0.80187218],
        [ 0.95813935,  0.87593263,  0.35781727,  0.50099513],
        [ 0.68346294,  0.71270203,  0.37025075,  0.56119619]])
```

With *avoid\_pure\_nash=True*:

```
>>> g = unit_vector_game(4, avoid_pure_nash=True, random_state=1234)
>>> g.players[0]
Player([[ 1.,  1.,  0.,  0.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  1.,  1.],
        [ 0.,  0.,  0.,  0.]])
>>> g.players[1]
Player([[ 0.19151945,  0.62210877,  0.43772774,  0.78535858],
        [ 0.77997581,  0.27259261,  0.27646426,  0.80187218],
        [ 0.95813935,  0.87593263,  0.35781727,  0.50099513],
        [ 0.68346294,  0.71270203,  0.37025075,  0.56119619]])
>>> pure_nash_brute(g)
[]
```



## 2.1 approximation

### 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:  $y_t = \bar{y} + \rho y_{t-1} + \varepsilon_t$  where  $\varepsilon_t$  is i.i.d. normal of mean 0, std dev of *sigma*

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

$$\theta_2 = \begin{bmatrix} p & 1-p \\ 1-q & q \end{bmatrix}$$

$$\theta_{n+1} = p \begin{bmatrix} \theta_n & 0 \\ 0 & 0 \end{bmatrix} + (1-p) \begin{bmatrix} 0 & \theta_n \\ 0 & 0 \end{bmatrix} + q \begin{bmatrix} 0 & 0 \\ \theta_n & 0 \end{bmatrix} + (1-q) \begin{bmatrix} 0 & 0 \\ 0 & \theta_n \end{bmatrix}$$

#### Parameters

**n** [int] The number of points to approximate the distribution

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

**sigma** [float] The value of the standard deviation of the  $\varepsilon$  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 i.i.d. 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

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  $i \rightarrow j$ , if  $i = j$  or  $P^k[i, j] > 0$  for some  $k = 1, 2, \dots$ ;  $i$  and  $j$  communicate, denoted  $i \leftrightarrow j$ , if  $i \rightarrow j$  and  $j \rightarrow i$ . 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  $i \rightarrow j$  implies  $j \rightarrow i$ ; 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 there is an edge from some node in  $C$  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  $P = \begin{bmatrix} 1 & 0 \\ 0 & 5 & 0.5 \end{bmatrix}$ . 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  $n \times n$ .
- 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

<code>get_index(value)</code>	Return the index (or indices) of the given value (or values) in <i>state_values</i> .
<code>simulate(ts_length[, init, num_reps, ...])</code>	Simulate time series of state transitions, where the states are annotated with their values (if <i>state_values</i> is not None).
<code>simulate_indices(ts_length[, init, ...])</code>	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 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  $k = \text{len}(\text{init})$  if (init, num\_reps) = (array, None),  $k = \text{num\_reps}$  if (init, num\_reps) = (int or None, int), and  $k = \text{len}(\text{init}) * \text{num\_reps}$  if (init, num\_reps) = (array, int).

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

**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  $S = \{0, \dots, n-1\}$ .

#### 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

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  $r: SA \rightarrow \mathbb{R}$ , where  $r(s, a)$  is the reward when the current state is  $s$  and the action chosen is  $a$ ;
- transition probability function  $q: SA \rightarrow \Delta(S)$ , 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  $\sigma$ , let  $r_\sigma$  and  $Q_\sigma$  be the reward vector and the transition probability matrix for  $\sigma$ , 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  $v_\sigma$  for  $\sigma$  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  $\sigma^*$  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  $\sigma$ , the operator  $T_\sigma$  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  $\sigma^*$  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  $\varepsilon$ -optimal policy and an  $\varepsilon/2$ -approximation of the optimal value function for a prespecified value of  $\varepsilon$ .

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  $\text{span}(Tv - v) < [(1 - \beta)/\beta]\varepsilon$  is satisfied, where  $\text{span}(z) = \max(z) - \min(z)$ .

## 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

<code>clear()</code>	
<code>copy()</code>	
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get(k[,d])</code>	
<code>items()</code>	
<code>keys()</code>	
<code>pop(k[,d])</code>	If key is not found, d is returned if given, otherwise <code>KeyError</code> is raised

Continued on next page

Table 2 – continued from previous page

<code>popitem()</code>	2-tuple; but raise <code>KeyError</code> if <code>D</code> is empty.
<code>setdefault(k,d)</code>	
<code>update([E, ]**F)</code>	If <code>E</code> is present and has a <code>.keys()</code> method, then does: for <code>k</code> in <code>E</code> : <code>D[k] = E[k]</code> If <code>E</code> is present and lacks a <code>.keys()</code> method, then does: for <code>k, v</code> in <code>E</code> : <code>D[k] = v</code> In either case, this is followed by: for <code>k</code> in <code>F</code> : <code>D[k] = F[k]</code>
<code>values()</code>	

**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 feasible 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  $q(s_{\text{next}}, a)$ :  
 $q(0|0, 0) = 0.5, q(1|0, 0) = 0.5, q(0|0, 1) = 0, q(1|0, 1) = 1, q(0|1, 0) = 0, q(1|1, 0) = 1$
- Discount factor 0.95

### Creating a 'DiscreteDP' instance

#### *Product formulation*

This approach uses the product set  $S \times A$  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.          ])
```

(continues on next page)

(continued from previous page)

```
>>> 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

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

Continued on next page

Table 3 – continued from previous page

<code>solve([method, v_init, epsilon, max_iter, k])</code>	Solve the dynamic programming problem.
<code>to_product_form()</code>	Convert this instance of <i>DiscreteDP</i> to the “product” form.
<code>to_sa_pair_form([sparse])</code>	Convert this instance of <i>DiscreteDP</i> to SA-pair form
<code>value_iteration([v_init, epsilon, max_iter])</code>	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, optional(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 represented as a DPSolveResult. See *DPSolveResult* for details.

**to\_product\_form()**

Convert this instance of *DiscreteDP* to the “product” form.

The product form uses the version of the `init` method taking *R*, *Q* and *beta*.

#### Returns

**ddp\_sa** [DiscreteDP] The corresponding DiscreteDP instance in product form

#### Notes

If this instance is already in product form then it is returned un-modified

**to\_sa\_pair\_form** (*sparse=True*)

Convert this instance of *DiscreteDP* to SA-pair form

#### Parameters

**sparse** [bool, optional(default=True)] Should the *Q* matrix be stored as a sparse matrix? If true the CSR format is used

#### Returns

**ddp\_sa** [DiscreteDP] The corresponding DiscreteDP instance in SA-pair form

#### Notes

If this instance is already in SA-pair form then it is returned un-modified

**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  $t = T, \dots, 1$ , 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  $t = 0, \dots, T$ .

**sigmas** [ndarray(int, ndim=2)] Array of shape  $(T, n)$  where  $sigmas[t]$  contains the optimal policy function at period  $t = 0, \dots, T-1$ .

## 2.4 gth\_solve

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 recurrent 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 [1], 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 [2], Chapter 10.

### Parameters

**A** [array\_like(float, ndim=2)] Stochastic matrix or generator matrix. Must be of shape  $n \times n$ .

### Returns

**x** [numpy.ndarray(float, ndim=1)] Stationary distribution of *A*.

**overwrite** [bool, optional(default=False)] Whether to overwrite *A*.

### References

[1], [2]

## 2.5 random

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])
```



### 3.1 utilities

Utilities to Support Random Operations and Generating Vectors and Matrices

`quantecon.random.utilities.draw`

Generate a random sample according to the cumulative distribution given by *cdf*. Jit-compiled by Numba in nopython mode.

#### Parameters

**cdf** [array\_like(float, ndim=1)] Array containing the cumulative distribution.

**size** [scalar(int), optional(default=None)] Size of the sample. If an integer is supplied, an ndarray of *size* independent draws is returned; otherwise, a single draw is returned as a scalar.

#### Returns

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

#### Examples

```
>>> cdf = np.cumsum([0.4, 0.6])
>>> qe.random.draw(cdf)
1
>>> qe.random.draw(cdf, 10)
array([1, 0, 1, 0, 1, 0, 0, 0, 1, 0])
```

`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]])
```

## 4.1 arma

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

$$X_t = \phi_1 X_{t-1} + \dots + \phi_p X_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \dots + \theta_q \epsilon_{t-q}$$

where

- $\phi = (\phi_1, \phi_2, \dots, \phi_p)$
- $\theta = (\theta_1, \theta_2, \dots, \theta_q)$
- $\sigma$  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 Parameters]

**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**

<code>autocovariance([num_autocov])</code>	Compute the autocovariance function from the ARMA parameters over the integers <code>range(num_autocov)</code> using the spectral density and the inverse Fourier transform.
<code>impulse_response([impulse_length])</code>	Get the impulse response corresponding to our model.
<code>set_params()</code>	Internally, <code>scipy.signal</code> works with systems of the form
<code>simulation([ts_length, random_state])</code>	Compute a simulated sample path assuming Gaussian shocks.
<code>spectral_density([two_pi, res])</code>	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**

**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

$$ar_{poly} = (1, -\phi_1, -\phi_2, \dots, -\phi_p)$$

$$ma_{poly} = (1, \theta_1, \theta_2, \dots, \theta_q)$$

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, random\_state=None*)

Compute a simulated sample path assuming Gaussian shocks.

#### Parameters

**ts\_length** [scalar(int), optional(default=90)] Number of periods to simulate for

**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

**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 ce\_util

Utility functions used in CompEcon

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

### 4.2.1 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*)

Expands one or more vectors (or matrices) into a matrix where rows span the cartesian product of combinations of the input arrays. Each column of the input arrays will correspond to one column of the output matrix.

### Parameters

**\*arrays** [tuple/list of np.ndarray] Tuple/list of vectors to be expanded.

### Returns

**out** [np.ndarray] The cartesian product of combinations of the input arrays.

## 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.

## 4.3 compute\_fp

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  $\|T^k v - T^{k-1} v\| \leq \text{error\_tol}$  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 [1], 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  $\|T(v) - v\| \leq \text{error\_tol}$ , provided  $T$  is a function that satisfies the assumptions of Brouwer’s fixed point theorem, 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**

[1]

## 4.4 discrete\_rv

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** [see Parameters.] Getter method for *q*.

**Q** [array\_like(float)] The cumulative sum of *q*.

**Methods**


---

<code>draw</code> ([ <i>k</i> , <i>random_state</i> ])	Returns <i>k</i> draws from <i>q</i> .
--	--

---

**draw** (*k=1*, *random\_state=None*)

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

**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**

**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()
```

## 4.5 distributions

Probability distributions useful in economics.

### 4.5.1 References

[http://en.wikipedia.org/wiki/Beta-binomial\\_distribution](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

---

<code>pdf()</code>	Generate the vector of probabilities for the Beta-binomial ( <i>n</i> , <i>a</i> , <i>b</i> ) distribution.
--------------------	---

---

#### **mean**

#### **pdf()**

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

The Beta-binomial distribution takes the form

$$p(k | n, a, b) = \binom{n}{k} \frac{B(k + a, n - k + b)}{B(a, b)}, \quad k = 0, \dots, n,$$

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

## 4.6 ecdf

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  $x$

---

## 4.7 estspec

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 frequencies at which periodogram is evaluated

**I\_w** [array\_like(float)] Values of periodogram at the Fourier frequencies

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

Computes the periodogram

$$I(w) = \frac{1}{n} \left[ \sum_{t=0}^{n-1} x_t e^{itw} \right]^2$$

at the Fourier frequencies  $w_j := \frac{2\pi j}{n}$ ,  $j = 0, \dots, n-1$ , using the fast Fourier transform. Only the frequencies  $w_j$  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 frequencies at which periodogram is evaluated

**I\_w** [array\_like(float)] Values of periodogram at the Fourier frequencies

`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.

## 4.8 graph\_tools

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 [1] and periodicity [2].

#### Parameters

**adj\_matrix** [array\_like(ndim=2)] Adjacency matrix representing a directed graph. Must be of shape  $n \times n$ .

**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

[1], [2]

#### 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

---

<code>subgraph(nodes)</code>	Return the subgraph consisting of the given nodes and edges between these 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 these 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*)

`quantecon.graph_tools.random_tournament_graph` (*n*, *random\_state=None*)

Return a random tournament graph [1] with *n* nodes.

#### Parameters

**n** [scalar(int)] Number of nodes.

**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

**DiGraph** A DiGraph representing the tournament graph.

## References

[1]

## 4.9 gridtools

Implements cartesian products and regular cartesian grids, and provides a function that constructs a grid for a simplex as well as one that determines the index of a point in the simplex.

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

Cartesian product of a list of arrays

### Parameters

**nodes** [list(array\_like(ndim=1))]

**order** [str, optional(default='C')] ('C' or 'F') order in which the product is enumerated

### Returns

**out** [ndarray(ndim=2)] 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** [array\_like(ndim=1)] lower bounds in each dimension

**b** [array\_like(ndim=1)] upper bounds in each dimension

**nums** [array\_like(ndim=1)] number of nodes along each dimension

**order** [str, optional(default='C')] ('C' or 'F') order in which the product is enumerated

### Returns

**out** [ndarray(ndim=2)] each line corresponds to one point of the product space

`quantecon.gridtools.num_compositions(m, n)`

The total number of m-part compositions of n, which is equal to  $(n+m-1) \text{ choose } (m-1)$ .

### Parameters

**m** [scalar(int)] Number of parts of composition.

**n** [scalar(int)] Integer to decompose.

### Returns

**scalar(int)** Total number of m-part compositions of n.

`quantecon.gridtools.num_compositions_jit`

Numba jit version of `num_compositions`. Return 0 if the outcome exceeds the maximum value of `np.intp`.

`quantecon.gridtools.simplex_grid`

Construct an array consisting of the integer points in the  $(m-1)$ -dimensional simplex  $\{x \mid x_0 + \dots + x_{m-1} = n\}$ , or equivalently, the m-part compositions of n, which are listed in lexicographic order. The total number of the points (hence the length of the output array) is  $L = (n+m-1)! / (n! * (m-1)!)$  (i.e.,  $(n+m-1) \text{ choose } (m-1)$ ).

### Parameters

**m** [scalar(int)] Dimension of each point. Must be a positive integer.

**n** [scalar(int)] Number which the coordinates of each point sum to. Must be a nonnegative integer.

### Returns

**out** [ndarray(int, ndim=2)] Array of shape  $(L, m)$  containing the integer points in the simplex, aligned in lexicographic order.

## Notes

A grid of the  $(m-1)$ -dimensional *unit* simplex with  $n$  subdivisions along each dimension can be obtained by `simplex_grid(m, n) / n`.

## References

A. Nijenhuis and H. S. Wilf, *Combinatorial Algorithms*, Chapter 5, Academic Press, 1978.

## Examples

```
>>> simplex_grid(3, 4)
array([[0, 0, 4],
       [0, 1, 3],
       [0, 2, 2],
       [0, 3, 1],
       [0, 4, 0],
       [1, 0, 3],
       [1, 1, 2],
       [1, 2, 1],
       [1, 3, 0],
       [2, 0, 2],
       [2, 1, 1],
       [2, 2, 0],
       [3, 0, 1],
       [3, 1, 0],
       [4, 0, 0]])
```

```
>>> simplex_grid(3, 4) / 4
array([[ 0. ,  0. ,  1. ],
       [ 0. ,  0.25,  0.75],
       [ 0. ,  0.5 ,  0.5 ],
       [ 0. ,  0.75,  0.25],
       [ 0. ,  1. ,  0. ],
       [ 0.25,  0. ,  0.75],
       [ 0.25,  0.25,  0.5 ],
       [ 0.25,  0.5 ,  0.25],
       [ 0.25,  0.75,  0. ],
       [ 0.5 ,  0. ,  0.5 ],
       [ 0.5 ,  0.25,  0.25],
       [ 0.5 ,  0.5 ,  0. ],
       [ 0.75,  0. ,  0.25],
       [ 0.75,  0.25,  0. ],
       [ 1. ,  0. ,  0. ]])
```

`quantecon.gridtools.simplex_index(x, m, n)`

Return the index of the point  $x$  in the lexicographic order of the integer points of the  $(m-1)$ -dimensional simplex  $\{x \mid x_0 + \dots + x_{m-1} = n\}$ .

### Parameters

**x** [array\_like(int, ndim=1)] Integer point in the simplex, i.e., an array of  $m$  nonnegative integers that sum to  $n$ .

**m** [scalar(int)] Dimension of each point. Must be a positive integer.

**n** [scalar(int)] Number which the coordinates of each point sum to. Must be a nonnegative integer.

**Returns**

**idx** [scalar(int)] Index of x.

## 4.10 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.

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

Bases: `scipy.integrate._ode.ode`

Creates an instance of the IVP class.

**Parameters**

**f** [callable `f(t, y, *f_args)`] Right hand side of the system of equations defining the ODE. The independent variable, `t`, is a scalar; `y` is an ndarray of dependent variables with `y.shape == (n,)`. The function `f` should return a scalar, ndarray or list (but not a tuple).

**jac** [callable `jac(t, y, *jac_args)`, optional(default=None)] Jacobian of the right hand side of the system of equations defining the ODE.

**Attributes**

`y`

**Methods**

<code>compute_residual(traj, ti[, k, ext])</code>	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.
<code>get_return_code()</code>	Extracts the return code for the integration to enable better control if the integration fails.
<code>integrate(t[, step, relax])</code>	Find <code>y=y(t)</code> , set <code>y</code> as an initial condition, and return <code>y</code> .
<code>interpolate(traj, ti[, k, der, ext])</code>	Parametric B-spline interpolation in N-dimensions.
<code>set_f_params(*args)</code>	Set extra parameters for user-supplied function <code>f</code> .
<code>set_initial_value(y[, t])</code>	Set initial conditions <code>y(t) = y</code> .
<code>set_integrator(name, **integrator_params)</code>	Set integrator by name.
<code>set_jac_params(*args)</code>	Set extra parameters for user-supplied function <code>jac</code> .
<code>set_solout(solout)</code>	Set callable to be called at every successful integration step.

Continued on next page



Table 6 – continued from previous page

<code>solve(t0, y0[, h, T, g, tol, integrator, ...])</code>	Solve the IVP by integrating the ODE given some initial condition.
<code>successful()</code>	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  $1 \leq k \leq 5$ .

**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 specified.
- 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.

## 4.11 kalman

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

### 4.11.1 References

<https://lectures.quantecon.org/py/kalman.html>

**class** quantecon.kalman.**Kalman** (*ss, x\_hat=None, Sigma=None*)  
Bases: `object`

Implements the Kalman filter for the Gaussian state space model

$$\begin{aligned}x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t + Hv_t\end{aligned}$$

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  $n \times 1$  array representing the mean  $\hat{x}$  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

<https://lectures.quantecon.org/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

<code>filtered_to_forecast()</code>	Updates the moments of the time t filtering distribution to the moments of the predictive distribution, which becomes the time t+1 prior
<code>prior_to_filtered(y)</code>	Updates the moments (x_hat, Sigma) of the time t prior to the time t filtering distribution, using current measurement $y_t$ .
<code>stationary_coefficients(j[, coeff_type])</code>	Wold representation moving average or VAR coefficients for the steady state Kalman filter.
<code>stationary_values()</code>	Computes the limit of $\Sigma_t$ as t goes to infinity by solving the associated Riccati equation.
<code>update(y)</code>	Updates x_hat and Sigma given k x 1 ndarray y.
<code>whitener_lss()</code>	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 representation given by

<b>set_state</b>	
<b>stationary_innovation_covar</b>	

## K\_infinity

## Sigma\_infinity

## 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

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

## 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 coefficient 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 representation given by

$$\tilde{x}_{t+1}^* = \tilde{A}\tilde{x} + \tilde{C}va = \tilde{G}\tilde{x}$$

where

$$\tilde{x}_t = [x + t, \hat{x}_t, v_t]$$

and

$$\tilde{A} = \begin{bmatrix} A & 0 & 0 \\ KG & A - KG & KH \\ 0 & 0 & 0 \end{bmatrix}$$

$$\tilde{C} = \begin{bmatrix} C & 0 \\ 0 & 0 \\ 0 & I \end{bmatrix}$$

$$\tilde{G} = [G \quad -G \quad H]$$

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

## 4.12 lae

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

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

This is a density in  $y$ .

### 4.12.1 References

[https://lectures.quantecon.org/py/stationary\\_densities.html](https://lectures.quantecon.org/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

---

<code>__call__(y)</code>	A vectorized function that returns the value of the look ahead estimate at the values in the array $y$ .
--------------------------	--

---

## 4.13 lqcontrol

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

$$\min \mathbb{E} \left[ \sum_{t=0}^{\infty} \beta^t r(x_t, u_t) \right]$$

with

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

or the finite horizon form

$$\min \mathbb{E} \left[ \sum_{t=0}^{T-1} \beta^t r(x_t, u_t) + \beta^T x_T' R_f x_T \right]$$

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  $n \times 1$ ,  $u$  is  $k \times 1$ ,  $w$  is  $j \times 1$  and the matrices are conformable for these dimensions. The sequence  $w_t$  is assumed to be white noise, with zero mean and  $\mathbb{E}[w_t' w_t] = I$ , the  $j \times j$  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  $k \times k$ . Should be symmetric and non-negative definite
- R** [array\_like(float)] R is the payoff (or cost) matrix that corresponds with the state variable  $x$  and is  $n \times n$ . Should be symmetric and non-negative definite
- A** [array\_like(float)] A is part of the state transition as described above. It should be  $n \times n$
- B** [array\_like(float)] B is part of the state transition as described above. It should be  $n \times k$
- 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
- N** [array\_like(float), optional(default=None)] N is the cross product term in the payoff, as above. It should be  $k \times n$ .
- 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  $n \times n$ . Should be symmetric 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  $V(x) = x' P x + d$
- d** [array\_like(float)] d is part of the value function representation of  $V(x) = x' P x + d$
- 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

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

**compute\_sequence** ( $x0$ ,  $ts\_length=None$ ,  $method='doubling'$ ,  $random\_state=None$ )

Compute and return the optimal state and control sequences  $x_0, \dots, x_T$  and  $u_0, \dots, u_T$  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

**method** [str, optional(default='doubling')] Solution method used in solving the associated Riccati equation, str in {'doubling', 'qz'}. Only relevant when the  $T$  attribute is *None* (i.e., the horizon is infinite).

**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\_path** [array\_like(float)] An  $n \times T+1$  matrix, where the  $t$ -th column represents  $x_t$

**u\_path** [array\_like(float)] A  $k \times T$  matrix, where the  $t$ -th column represents  $u_t$

**w\_path** [array\_like(float)] A  $j \times T+1$  matrix, where the  $t$ -th column represent  $w_t$

**stationary\_values** ( $method='doubling'$ )

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

$$V(x) = x'Px + d$$

in the infinite horizon case. Also computes the control matrix  $F$  from  $u = -Fx$ . Computation is via the solution algorithm as specified by the *method* option (default to the doubling algorithm) (see the documentation in *matrix\_eqn.solve\_discrete\_riccati*).

### Parameters

**method** [str, optional(default='doubling')] Solution method used in solving the associated Riccati equation, str in {'doubling', 'qz'}.

### Returns

**P** [array\_like(float)]  $P$  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.

**d** [array\_like(float)]  $d$  is part of the value function representation of  $V(x) = x'Px + d$

**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  $F_{t-1}$

## 4.14 lqnash

`quantecon.lqnash.nnash` ( $A, B1, B2, R1, R2, Q1, Q2, S1, S2, W1, W2, M1, M2, beta=1.0, tol=1e-08, max\_iter=1000, random\_state=None$ )

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 iterations allowed

**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

**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

## 4.15 lss

Computes quantities associated with the Gaussian linear state space model.

### 4.15.1 References

[https://lectures.quantecon.org/py/linear\\_models.html](https://lectures.quantecon.org/py/linear_models.html)

**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:

$$\begin{aligned}x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t + Hv_t\end{aligned}$$

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  $x_0 \sim N(\mu_0, \Sigma_0)$ . 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  $n \times n$
- C** [array\_like or scalar(float)] Part of the state transition equation. It should be  $n \times m$
- G** [array\_like or scalar(float)] Part of the observation equation. It should be  $k \times n$
- H** [array\_like or scalar(float), optional(default=None)] Part of the observation equation. It should be  $k \times l$
- mu\_0** [array\_like or scalar(float), optional(default=None)] This is the mean of initial draw and is  $n \times l$
- Sigma\_0** [array\_like or scalar(float), optional(default=None)] This is the variance of the initial draw and is  $n \times n$  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

<code>convert(x)</code>	Convert array_like objects (lists of lists, floats, etc.) into well formed 2D NumPy arrays
<code>geometric_sums(beta, x_t)</code>	Forecast the geometric sums
<code>impulse_response(lj)</code>	Pulls off the impulse response coefficients to a shock in $w_t$ for $x$ and $y$

Continued on next page

Table 10 – continued from previous page

<code>moment_sequence()</code>	Create a generator to calculate the population mean and variance-covariance matrix for both $x_t$ and $y_t$ starting at the initial condition (self.mu_0, self.Sigma_0).
<code>replicate([T, num_reps, random_state])</code>	Simulate num_reps observations of $x_T$ and $y_T$ given $x_0 \sim N(\mu_0, \Sigma_0)$ .
<code>simulate([ts_length, random_state])</code>	Simulate a time series of length ts_length, first drawing
<code>stationary_distributions([max_iter, tol])</code>	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

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

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

**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 impulse 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  $C, AC, A^2C...$
- $y$  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-covariance 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  $n \times 1$  array representing the population mean of  $x_t$

**mu\_y** [array\_like(float)] A  $k \times 1$  array representing the population mean of  $y_t$

**Sigma\_x** [array\_like(float)] An  $n \times n$  array representing the variance-covariance matrix of  $x_t$

**Sigma\_y** [array\_like(float)] A  $k \times k$  array representing the variance-covariance matrix of  $y_t$

**replicate** ( $T=10$ ,  $num\_reps=100$ ,  $random\_state=None$ )

Simulate  $num\_reps$  observations of  $x_T$  and  $y_T$  given  $x_0 \sim N(\mu_0, \Sigma_0)$ .

#### 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

**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(float)] An  $n \times num\_reps$  array, where the  $j$ -th column is the  $j$ -th observation of  $x_T$

**y** [array\_like(float)] A  $k \times num\_reps$  array, where the  $j$ -th column is the  $j$ -th observation of  $y_T$

**simulate** ( $ts\_length=100$ ,  $random\_state=None$ )

Simulate a time series of length  $ts\_length$ , first drawing

$$x_0 \sim N(\mu_0, \Sigma_0)$$

#### Parameters

**ts\_length** [scalar(int), optional(default=100)] The length of the 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** [array\_like(float)] An  $n \times ts\_length$  array, where the  $t$ -th column is  $x_t$

**y** [array\_like(float)] A  $k \times ts\_length$  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  $n \times 1$  array representing the stationary mean of  $x_t$

**mu\_y\_star** [array\_like(float)] An  $k \times 1$  array representing the stationary mean of  $y_t$

**Sigma\_x\_star** [array\_like(float)] An  $n \times n$  array representing the stationary var-cov matrix of  $x_t$

**Sigma\_y\_star** [array\_like(float)] An  $k \times k$  array representing the stationary var-cov matrix of  $y_t$

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

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,  $(m, n, k)$ ,  $m \times n \times k$  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.

**check\_valid** [{ 'warn', 'raise', 'ignore' }, optional] Behavior when the covariance matrix is not positive semidefinite.

**tol** [float, optional] Tolerance when checking the singular values in covariance matrix.

### 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,  $X = [x_1, x_2, \dots, x_N]$ . 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')
```

(continues on next page)

(continued from previous page)

```
>>> 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

[1],[2]

## 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

$$x_{t+1} = Ax_t + v_t$$

given  $x_0 = x_0$

Here  $x_t$  and  $v_t$  are both  $n \times 1$  and  $A$  is  $n \times n$ .

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  $n \times n$
- x0** [array\_like] Should be  $n \times 1$ . Initial condition
- v** [np.ndarray] Should be  $n \times ts\_length-1$ . 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  $x_t$

## 4.16 matrix\_eqn

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

- Lyapunov Equations

- Riccati 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  $j = 1, 2, \dots$  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  $n \times n$  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  $n \times n$  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  $X$

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

Solves the discrete-time algebraic Riccati equation

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

Computation is via a modified structured doubling algorithm, an explanation of which can be found in the reference below, if `method="doubling"` (default), and via a QZ decomposition method by calling `scipy.linalg.solve_discrete_are` if `method="qz"`.

### Parameters

- A** [array\_like(float, ndim=2)]  $k \times k$  array.
- B** [array\_like(float, ndim=2)]  $k \times n$  array
- Q** [array\_like(float, ndim=2)]  $k \times k$ , should be symmetric and non-negative definite
- R** [array\_like(float, ndim=2)]  $n \times n$ , should be symmetric and positive definite
- N** [array\_like(float, ndim=2)]  $n \times k$  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
- 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 "qz" then it uses `scipy.linalg.solve_discrete_are` (in which case `tolerance` and `max_iter` are irrelevant).

**Returns**

**X** [array\_like(float, ndim=2)] The fixed point of the Riccati equation; a  $k \times k$  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.

## 4.17 quad

Defining various quadrature routines.

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

### 4.17.1 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 Gauss-Chebyshev 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` ( $n, a, b, kind='N', equidist\_pp=None, random\_state=None$ )

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

**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

**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*, *usesqrm=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.qnwttrap` (*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 `qnwttrap` 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  $d \times d$  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` ( $n, a=1.0, b=1.0, tol=3e-14$ )

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
- a** [scalar or array\_like(float)] Shape parameter of the gamma distribution parameter. Must be positive
- b** [scalar or array\_like(float)] Scale parameter of the gamma distribution parameter. Must be positive
- tol** [scalar or array\_like(float)] Tolerance parameter for newton iterations for each node

### 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.

## 4.18 quadsums

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  $n \times n$  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  $n \times n$  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

$$q(x_0) = \mathbb{E} \left[ \sum_{t=0}^{\infty} \beta^t x_t' H x_t \right]$$

Here  $x_t$  is the VAR process  $x_{t+1} = Ax_t + Cw_t$  with  $x_t$  standard normal and  $x_0$  the initial condition.

### Parameters

- A** [array\_like(float, ndim=2)] The matrix described above in description. Should be  $n \times n$
- C** [array\_like(float, ndim=2)] The matrix described above in description. Should be  $n \times n$
- H** [array\_like(float, ndim=2)] The matrix described above in description. Should be  $n \times n$
- beta:** scalar(float) Should take a value in  $(0, 1)$
- x\_0:** array\_like(float, ndim=1) The initial condition. 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  $Q = H + \beta A'QA$ , and
- $v = \frac{\text{trace}(C'QC)\beta}{(1-\beta)}$

## 4.19 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.

## 4.20 robustlq

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

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

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

<code>F_to_K(F)</code>	Compute agent 2's best cost-minimizing response K, given F.
<code>K_to_F(K)</code>	Compute agent 1's best value-maximizing response F, given K.
<code>b_operator(P)</code>	The B operator, mapping P into
<code>compute_deterministic_entropy(F, K, x0)</code>	Given K and F, compute the value of deterministic entropy, which is
<code>d_operator(P)</code>	The D operator, mapping P into
<code>evaluate_F(F)</code>	Given a fixed policy F, with the interpretation $u = -Fx$ , this function computes the matrix $P_F$ and constant $d_F$ associated with discounted cost $J_F(x) = x' P_F x + d_F$

Continued on next page

Table 11 – continued from previous page

<code>robust_rule()</code>	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
<code>robust_rule_simple([P_init, max_iter, tol])</code>	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  $k \times n$  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  $j \times n$  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

$$B(P) := R - \beta^2 A' P B (Q + \beta B' P B)^{-1} B' P A + \beta A' P A$$

and also returning

$$F := (Q + \beta B' P B)^{-1} \beta B' P A$$

**Parameters**

$P$  [array\_like(float, ndim=2)] A matrix that should be  $n \times n$

**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  $k \times n$  array
- K** [array\_like(float, ndim=2)] The worst case matrix, a  $j \times n$  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

$$D(P) := P + PC(\theta I - C'PC)^{-1}C'P.$$

**Parameters**

- P** [array\_like(float, ndim=2)] A matrix that should be  $n \times n$

**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  $u = -Fx$ , this function computes the matrix  $P_F$  and constant  $d_F$  associated with discounted cost  $J_F(x) = x'P_Fx + d_F$

**Parameters**

- F** [array\_like(float, ndim=2)] The policy function, a  $k \times n$  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

## 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  $a[i-1] \leq v < a[i]$  (for  $i = 0$ ,  $v < a[0]$ ); if  $v[n-1] \leq v$ , return  $n$ , where  $n = \text{len}(a)$ .

#### 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  $a[i-1] \leq v < a[i]$ , or  $\text{len}(a)$  if no such index exists.

#### Notes

This routine is jit-compiled if the module Numba is available; 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 combinatorics

Useful routines for combinatorics

`quantecon.util.combinatorics.k_array_rank(a)`

Given an array *a* of *k* distinct nonnegative integers, sorted in ascending order, return its ranking in the lexicographic ordering of the descending sequences of the elements [1].

### Parameters

**a** [ndarray(int, ndim=1)] Array of length *k*.

### Returns

**idx** [scalar(int)] Ranking of *a*.

### References

[1]

`quantecon.util.combinatorics.k_array_rank_jit`

Numba jit version of *k\_array\_rank*.

### Notes

An incorrect value will be returned without warning or error if overflow occurs during the computation. It is the user's responsibility to ensure that the rank of the input array fits within the range of possible values of *np.intp*; a sufficient condition for it is *scipy.special.comb(a[-1]+1, len(a), exact=True) <= np.iinfo(np.intp).max*.

`quantecon.util.combinatorics.next_k_array`

Given an array *a* of *k* distinct nonnegative integers, sorted in ascending order, return the next *k*-array in the lexicographic ordering of the descending sequences of the elements [1]. *a* is modified in place.

### Parameters

**a** [ndarray(int, ndim=1)] Array of length *k*.

### Returns

**a** [ndarray(int, ndim=1)] View of *a*.

### References

[1]

## Examples

Enumerate all the subsets with  $k$  elements of the set  $\{0, \dots, n-1\}$ .

```
>>> n, k = 4, 2
>>> a = np.arange(k)
>>> while a[-1] < n:
...     print(a)
...     a = next_k_array(a)
...
[0 1]
[0 2]
[1 2]
[0 3]
[1 3]
[2 3]
```

## 5.3 common\_messages

### 5.3.1 Warnings Module

Contains a collection of warning messages for consistent package wide notifications

## 5.4 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.

### 5.4.1 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`

### 5.4.2 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', over-
write=False, verbose=True)
```

Retrieve raw files from QuantEcon.notebooks or 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(default=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)

### Examples

Consider a notebook that is dependant on a csv file to execute. If this file is located in a Github repository then it can be fetched using this utility

Assuming the file is at the root level in the master branch then:

```
>>> from quantecon.util import fetch_nb_dependencies
>>> status = fetch_nb_dependencies(["test.csv"], repo="https://<github_address>")
```

More than one file may be requested in the list provided

```
>>> status = fetch_nb_dependencies(["test.csv", "data.csv"], repo="https://
↳<github_address>")
```

A folder location can be added using folder=

```
>>> status = fetch_nb_dependencies("test.csv", report="https://<github_address>",
↳folder="data")
```

You can also specify a specific branch using branch= keyword argument.

This will download the requested file(s) to your local working directory. The default behaviour is **not** to overwrite a local file if it is present. This can be switched off by setting `overwrite=True`.

## 5.5 numba

Utilities to support Numba jitted functions

`quantecon.util.numba.comb_jit`

Numba jitted function that computes  $N$  choose  $k$ . Return 0 if the outcome exceeds the maximum value of `np.intp` or if  $N < 0$ ,  $k < 0$ , or  $k > N$ .

### Parameters

**N** [scalar(int)]

**k** [scalar(int)]

### Returns

**val** [scalar(int)]

## 5.6 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.

## 5.7 timing

Provides Matlab-like tic, tac and toc functions.

`quantecon.util.timing.loop_timer(n, function, args=None, verbose=True, digits=2, best_of=3)`

Return and print the total and average time elapsed for n runs of function.

### Parameters

**n** [scalar(int)] Number of runs.

**function** [function] Function to be timed.

**args** [list, optional(default=None)] Arguments of the function.

**verbose** [bool, optional(default=True)] If True, then prints average time.

**digits** [scalar(int), optional(default=2)] Number of digits printed for time elapsed.

**best\_of** [scalar(int), optional(default=3)] Average time over best\_of runs.

### Returns

**average\_time** [scalar(float)] Average time elapsed for n runs of function.

**average\_of\_best** [scalar(float)] Average of best\_of times for n runs of function.

`quantecon.util.timing.tac(verbose=True, digits=2)`

Return and print elapsed time since last `tic()`, `tac()`, or `toc()`.

### Parameters

**verbose** [bool, optional(default=True)] If True, then prints time.

**digits** [scalar(int), optional(default=2)] Number of digits printed for time elapsed.

### Returns

**elapsed** [scalar(float)] Time elapsed since last `tic()`, `tac()`, or `toc()`.

`quantecon.util.timing.tic()`

Save time for future use with `tac()` or `toc()`.

`quantecon.util.timing.toc(verbose=True, digits=2)`

Return and print time elapsed since last `tic()`.

### Parameters

**verbose** [bool, optional(default=True)] If True, then prints time.

**digits** [scalar(int), optional(default=2)] Number of digits printed for time elapsed.

### Returns

**elapsed** [scalar(float)] Time elapsed since last *tic()*.



## CHAPTER 6

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



---

## Bibliography

---

- [1] B. Codenotti, S. De Rossi, and M. Pagan, “An Experimental Analysis of Lemke-Howson Algorithm,” arXiv:0811.3247, 2008.
- [2] C. E. Lemke and J. T. Howson, “Equilibrium Points of Bimatrix Games,” *Journal of the Society for Industrial and Applied Mathematics* (1964), 413-423.
- [3] 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.
- [1] A. McLennan and R. Tourky, “From Imitation Games to Kakutani,” 2006.
- [1] Y. Rinott and M. Scarsini, “On the Number of Pure Strategy Nash Equilibria in Random Games,” *Games and Economic Behavior* (2000), 274-293.
- [1] [Combinatorial number system](#), Wikipedia.
- [1] W. K. Grassmann, M. I. Taksar and D. P. Heyman, “Regenerative Analysis and Steady State Distributions for Markov Chains,” *Operations Research* (1985), 1107-1116.
- [2] W. J. Stewart, *Probability, Markov Chains, Queues, and Simulation*, Princeton University Press, 2009.
- [1] A. McLennan and R. Tourky, “From Imitation Games to Kakutani,” 2006.
- [1] [Strongly connected component](#), Wikipedia.
- [2] [Aperiodic graph](#), Wikipedia.
- [1] [Tournament \(graph theory\)](#), Wikipedia.
- [1] Papoulis, A., “*Probability, Random Variables, and Stochastic Processes*,” 3rd ed., New York: McGraw-Hill, 1991.
- [2] Duda, R. O., Hart, P. E., and Stork, D. G., “*Pattern Classification*,” 2nd ed., New York: Wiley, 2001.
- [1] [Combinatorial number system](#), Wikipedia.
- [1] [Combinatorial number system](#), Wikipedia.



**q**

quantecon.arma, 39  
quantecon.ce\_util, 41  
quantecon.compute\_fp, 42  
quantecon.discrete\_rv, 43  
quantecon.distributions, 45  
quantecon.ecdf, 46  
quantecon.estspec, 46  
quantecon.game\_theory.game\_generators.bimatrix\_generators, 15  
quantecon.game\_theory.lemke\_howson, 3  
quantecon.game\_theory.mclennan\_tourky, 4  
quantecon.game\_theory.normal\_form\_game, 6  
quantecon.game\_theory.pure\_nash, 12  
quantecon.game\_theory.random, 12  
quantecon.game\_theory.support\_enumeration, 13  
quantecon.game\_theory.utilities, 14  
quantecon.game\_theory.vertex\_enumeration, 15  
quantecon.graph\_tools, 47  
quantecon.gridtools, 50  
quantecon.ivp, 52  
quantecon.kalman, 54  
quantecon.lae, 57  
quantecon.lqcontrol, 57  
quantecon.lqnash, 60  
quantecon.lss, 61  
quantecon.markov.approximation, 21  
quantecon.markov.core, 22  
quantecon.markov.ddp, 26  
quantecon.markov.gth\_solve, 34  
quantecon.markov.random, 34  
quantecon.markov.utilities, 36  
quantecon.matrix\_eqn, 65  
quantecon.quad, 67  
quantecon.quadsums, 73  
quantecon.random.utilities, 37  
quantecon.rank\_nullspace, 74  
quantecon.robustlq, 75  
quantecon.util.array, 79  
quantecon.util.combinatorics, 80  
quantecon.util.common\_messages, 81  
quantecon.util.notebooks, 81  
quantecon.util.numba, 82  
quantecon.util.random, 83  
quantecon.util.timing, 83



**A**

annotate\_nodes() (in module quantecon.graph\_tools), 49  
 ar\_periodogram() (in module quantecon.estspec), 46  
 ARMA (class in quantecon.arma), 39  
 autocovariance() (quantecon.arma.ARMA method), 40

**B**

b\_operator() (quantecon.robustlq.RBLQ method), 76  
 backward\_induction() (in module quantecon.markov.ddp), 33  
 bellman\_operator() (quantecon.markov.ddp.DiscreteDP method), 31  
 best\_response() (quantecon.game\_theory.normal\_form\_game.Player method), 9  
 best\_response\_2p (in module quantecon.game\_theory.normal\_form\_game), 11  
 BetaBinomial (class in quantecon.distributions), 45  
 blotto\_game() (in module quantecon.game\_theory.game\_generators.bimatrix\_generators), 16

**C**

cartesian() (in module quantecon.gridtools), 50  
 cdfs (quantecon.markov.core.MarkovChain attribute), 24  
 cdfs1d (quantecon.markov.core.MarkovChain attribute), 24  
 check\_random\_state() (in module quantecon.util.random), 83  
 ckron() (in module quantecon.ce\_util), 41  
 comb\_jit (in module quantecon.util.numba), 82  
 communication\_classes (quantecon.markov.core.MarkovChain attribute), 24  
 communication\_classes\_indices (quantecon.markov.core.MarkovChain attribute), 24  
 compute\_deterministic\_entropy() (quantecon.robustlq.RBLQ method), 76

compute\_fixed\_point() (in module quantecon.compute\_fp), 42  
 compute\_greedy() (quantecon.markov.ddp.DiscreteDP method), 31  
 compute\_residual() (quantecon.ivp.IVP method), 53  
 compute\_sequence() (quantecon.lqcontrol.LQ method), 59  
 controlled\_mc() (quantecon.markov.ddp.DiscreteDP method), 31  
 convert() (quantecon.lss.LinearStateSpace method), 62  
 covariance\_game() (in module quantecon.game\_theory.random), 12  
 cyclic\_classes (quantecon.markov.core.MarkovChain attribute), 24  
 cyclic\_classes\_indices (quantecon.markov.core.MarkovChain attribute), 24  
 cyclic\_components (quantecon.graph\_tools.DiGraph attribute), 49  
 cyclic\_components\_indices (quantecon.graph\_tools.DiGraph attribute), 49

**D**

d\_operator() (quantecon.robustlq.RBLQ method), 77  
 DiGraph (class in quantecon.graph\_tools), 47  
 digraph (quantecon.markov.core.MarkovChain attribute), 24  
 DiscreteDP (class in quantecon.markov.ddp), 28  
 DiscreteRV (class in quantecon.discrete\_rv), 43  
 dominated\_actions() (quantecon.game\_theory.normal\_form\_game.Player method), 10  
 DPSolveResult (class in quantecon.markov.ddp), 27  
 draw (in module quantecon.random.utilities), 37  
 draw() (quantecon.discrete\_rv.DiscreteRV method), 43

**E**

ECDF (class in quantecon.ecdf), 46  
 evaluate\_F() (quantecon.robustlq.RBLQ method), 77

evaluate\_policy() (quantecon.markov.ddp.DiscreteDP method), 32

## F

F\_to\_K() (quantecon.robustlq.RBLQ method), 76

fetch\_nb\_dependencies() (in module quantecon.util.notebooks), 81

filtered\_to\_forecast() (quantecon.kalman.Kalman method), 55

## G

geometric\_sums() (quantecon.lss.LinearStateSpace method), 62

get\_index() (quantecon.markov.core.MarkovChain method), 24

gridmake() (in module quantecon.ce\_util), 42

gth\_solve() (in module quantecon.markov.gth\_solve), 34

## I

impulse\_response() (quantecon.arma.ARMA method), 40

impulse\_response() (quantecon.lss.LinearStateSpace method), 62

interpolate() (quantecon.ivp.IVP method), 53

is\_aperiodic (quantecon.graph\_tools.DiGraph attribute), 49

is\_aperiodic (quantecon.markov.core.MarkovChain attribute), 24

is\_best\_response() (quantecon.game\_theory.normal\_form\_game.Player method), 10

is\_dominated() (quantecon.game\_theory.normal\_form\_game.Player method), 10

is\_irreducible (quantecon.markov.core.MarkovChain attribute), 24

is\_nash() (quantecon.game\_theory.normal\_form\_game.NormalFormGame method), 8

is\_strongly\_connected (quantecon.graph\_tools.DiGraph attribute), 49

IVP (class in quantecon.ivp), 52

## K

k\_array\_rank() (in module quantecon.util.combinatorics), 80

k\_array\_rank\_jit (in module quantecon.util.combinatorics), 80

K\_infinity (quantecon.kalman.Kalman attribute), 55

K\_to\_F() (quantecon.robustlq.RBLQ method), 76

Kalman (class in quantecon.kalman), 54

## L

LAE (class in quantecon.lae), 57

lemke\_howson() (in module quantecon.game\_theory.lemke\_howson), 3

LinearStateSpace (class in quantecon.lss), 61

loop\_timer() (in module quantecon.util.timing), 83

LQ (class in quantecon.lqcontrol), 57

## M

m\_quadratic\_sum() (in module quantecon.quadsums), 73

MarkovChain (class in quantecon.markov.core), 23

mc\_compute\_stationary() (in module quantecon.markov.core), 25

mc\_sample\_path() (in module quantecon.markov.core), 25

mclennan\_tourky() (in module quantecon.game\_theory.mclennan\_tourky), 4

mean (quantecon.distributions.BetaBinomial attribute), 45

mlinspace() (in module quantecon.gridtools), 50

modified\_policy\_iteration() (quantecon.markov.ddp.DiscreteDP method), 32

moment\_sequence() (quantecon.lss.LinearStateSpace method), 62

multivariate\_normal() (in module quantecon.lss), 64

## N

NashResult (class in quantecon.game\_theory.utilities), 14

next\_k\_array (in module quantecon.util.combinatorics), 80

nnash() (in module quantecon.lqnash), 60

node\_labels (quantecon.graph\_tools.DiGraph attribute), 49

NormalFormGame (class in quantecon.game\_theory.normal\_form\_game), 8

nullspace() (in module quantecon.rank\_nullspace), 74

num\_communication\_classes (quantecon.markov.core.MarkovChain attribute), 24

num\_transitions() (in module quantecon.gridtools), 50

num\_compositions\_jit (in module quantecon.gridtools), 50

num\_recurrent\_classes (quantecon.markov.core.MarkovChain attribute), 24

num\_sink\_strongly\_connected\_components (quantecon.graph\_tools.DiGraph attribute), 49

num\_strongly\_connected\_components (quantecon.graph\_tools.DiGraph attribute), 49

## O

operator\_iteration() (quantecon.markov.ddp.DiscreteDP method), 32

## P

payoff\_profile\_array (quantecon.game\_theory.normal\_form\_game.NormalFormGame attribute), 8



- payoff\_vector() (quantecon.game\_theory.normal\_form\_game.Player method), 10  
 pdf() (quantecon.distributions.BetaBinomial method), 45  
 period (quantecon.graph\_tools.DiGraph attribute), 49  
 period (quantecon.markov.core.MarkovChain attribute), 24  
 periodogram() (in module quantecon.estspec), 47  
 phi (quantecon.arma.ARMA attribute), 40  
 Player (class in quantecon.game\_theory.normal\_form\_game), 8  
 policy\_iteration() (quantecon.markov.ddp.DiscreteDP method), 32  
 prior\_to\_filtered() (quantecon.kalman.Kalman method), 55  
 probvec() (in module quantecon.random.utilities), 37  
 pure2mixed() (in module quantecon.game\_theory.normal\_form\_game), 11  
 pure\_nash\_brute() (in module quantecon.game\_theory.pure\_nash), 12  
 pure\_nash\_brute\_gen() (in module quantecon.game\_theory.pure\_nash), 12
- ## Q
- q (quantecon.discrete\_rv.DiscreteRV attribute), 44  
 qnwbeta() (in module quantecon.quad), 72  
 qnwcheb() (in module quantecon.quad), 67  
 qnwequi() (in module quantecon.quad), 67  
 qnwgamma() (in module quantecon.quad), 72  
 qnwlege() (in module quantecon.quad), 68  
 qnwlogn() (in module quantecon.quad), 69  
 qnwnorm() (in module quantecon.quad), 69  
 qnwsimp() (in module quantecon.quad), 70  
 qnwtrap() (in module quantecon.quad), 70  
 qnwunif() (in module quantecon.quad), 71  
 quadrect() (in module quantecon.quad), 71  
 quantecon.arma (module), 39  
 quantecon.ce\_util (module), 41  
 quantecon.compute\_fp (module), 42  
 quantecon.discrete\_rv (module), 43  
 quantecon.distributions (module), 45  
 quantecon.ecdf (module), 46  
 quantecon.estspec (module), 46  
 quantecon.game\_theory.game\_generators.bimatrix\_generators (module), 15  
 quantecon.game\_theory.lemke\_howson (module), 3  
 quantecon.game\_theory.mclennan\_tourky (module), 4  
 quantecon.game\_theory.normal\_form\_game (module), 6  
 quantecon.game\_theory.pure\_nash (module), 12  
 quantecon.game\_theory.random (module), 12  
 quantecon.game\_theory.support\_enumeration (module), 13  
 quantecon.game\_theory.utilities (module), 14  
 quantecon.game\_theory.vertex\_enumeration (module), 15  
 quantecon.graph\_tools (module), 47  
 quantecon.gridtools (module), 50  
 quantecon.ivp (module), 52  
 quantecon.kalman (module), 54  
 quantecon.lae (module), 57  
 quantecon.lqcontrol (module), 57  
 quantecon.lqnash (module), 60  
 quantecon.lss (module), 61  
 quantecon.markov.approximation (module), 21  
 quantecon.markov.core (module), 22  
 quantecon.markov.ddp (module), 26  
 quantecon.markov.gth\_solve (module), 34  
 quantecon.markov.random (module), 34  
 quantecon.markov.utilities (module), 36  
 quantecon.matrix\_eqn (module), 65  
 quantecon.quad (module), 67  
 quantecon.quadsums (module), 73  
 quantecon.random.utilities (module), 37  
 quantecon.rank\_nullspace (module), 74  
 quantecon.robustlq (module), 75  
 quantecon.util.array (module), 79  
 quantecon.util.combinatorics (module), 80  
 quantecon.util.common\_messages (module), 81  
 quantecon.util.notebooks (module), 81  
 quantecon.util.numba (module), 82  
 quantecon.util.random (module), 83  
 quantecon.util.timing (module), 83
- ## R
- random\_choice() (quantecon.game\_theory.normal\_form\_game.Player method), 11  
 random\_discrete\_dp() (in module quantecon.markov.random), 34  
 random\_game() (in module quantecon.game\_theory.random), 13  
 random\_markov\_chain() (in module quantecon.markov.random), 35  
 random\_stochastic\_matrix() (in module quantecon.markov.random), 35  
 random\_tournament\_graph() (in module quantecon.graph\_tools), 49  
 rank\_est() (in module quantecon.rank\_nullspace), 74  
 ranking\_game() (in module quantecon.game\_theory.game\_generators.bimatrix\_generators), 17  
 RBLQ (class in quantecon.robustlq), 75  
 recurrent\_classes (quantecon.markov.core.MarkovChain attribute), 24  
 recurrent\_classes\_indices (quantecon.markov.core.MarkovChain attribute), 24

replicate() (quantecon.lss.LinearStateSpace method), 63  
 robust\_rule() (quantecon.robustlq.RBLQ method), 77  
 robust\_rule\_simple() (quantecon.robustlq.RBLQ method), 77  
 rouwenhorst() (in module quantecon.markov.approximation), 21  
 RQ\_sigma() (quantecon.markov.ddp.DiscreteDP method), 31

## S

sa\_indices (in module quantecon.markov.utilities), 36  
 sample\_without\_replacement (in module quantecon.random.utilities), 38  
 scc\_proj (quantecon.graph\_tools.DiGraph attribute), 49  
 searchsorted (in module quantecon.util.array), 79  
 set\_params() (quantecon.arma.ARMA method), 40  
 set\_state() (quantecon.kalman.Kalman method), 56  
 sgc\_game() (in module quantecon.game\_theory.game\_generators.bimatrix\_generators), 18  
 Sigma\_infinity (quantecon.kalman.Kalman attribute), 55  
 simplex\_grid (in module quantecon.gridtools), 50  
 simplex\_index() (in module quantecon.gridtools), 51  
 simulate() (quantecon.lss.LinearStateSpace method), 63  
 simulate() (quantecon.markov.core.MarkovChain method), 24  
 simulate\_indices() (quantecon.markov.core.MarkovChain method), 25  
 simulate\_linear\_model (in module quantecon.lss), 65  
 simulation() (quantecon.arma.ARMA method), 40  
 sink\_scc\_labels (quantecon.graph\_tools.DiGraph attribute), 49  
 sink\_strongly\_connected\_components (quantecon.graph\_tools.DiGraph attribute), 49  
 sink\_strongly\_connected\_components\_indices (quantecon.graph\_tools.DiGraph attribute), 49  
 skew (quantecon.distributions.BetaBinomial attribute), 46  
 smooth() (in module quantecon.estspect), 47  
 solve() (quantecon.ivp.IVP method), 53  
 solve() (quantecon.markov.ddp.DiscreteDP method), 32  
 solve\_discrete\_lyapunov() (in module quantecon.matrix\_eqn), 66  
 solve\_discrete\_riccati() (in module quantecon.matrix\_eqn), 66  
 spectral\_density() (quantecon.arma.ARMA method), 41  
 state\_values (quantecon.markov.core.MarkovChain attribute), 25  
 stationary\_coefficients() (quantecon.kalman.Kalman method), 56  
 stationary\_distributions (quantecon.markov.core.MarkovChain attribute), 25  
 stationary\_distributions() (quantecon.lss.LinearStateSpace method), 63

stationary\_innovation\_covar() (quantecon.kalman.Kalman method), 56  
 stationary\_values() (quantecon.kalman.Kalman method), 56  
 stationary\_values() (quantecon.lqcontrol.LQ method), 59  
 std (quantecon.distributions.BetaBinomial attribute), 46  
 std\_norm\_cdf (in module quantecon.markov.approximation), 21  
 strongly\_connected\_components (quantecon.graph\_tools.DiGraph attribute), 49  
 strongly\_connected\_components\_indices (quantecon.graph\_tools.DiGraph attribute), 49  
 subgraph() (quantecon.graph\_tools.DiGraph method), 49  
 support\_enumeration() (in module quantecon.game\_theory.support\_enumeration), 13  
 support\_enumeration\_gen() (in module quantecon.game\_theory.support\_enumeration), 13

## T

T\_sigma() (quantecon.markov.ddp.DiscreteDP method), 31  
 tac() (in module quantecon.util.timing), 83  
 tauchen() (in module quantecon.markov.approximation), 21  
 theta (quantecon.arma.ARMA attribute), 41  
 tic() (in module quantecon.util.timing), 83  
 to\_product\_form() (quantecon.markov.ddp.DiscreteDP method), 33  
 to\_sa\_pair\_form() (quantecon.markov.ddp.DiscreteDP method), 33  
 toc() (in module quantecon.util.timing), 83  
 tournament\_game() (in module quantecon.game\_theory.game\_generators.bimatrix\_generators), 18

## U

uniform() (in module quantecon.discrete\_rv), 44  
 unit\_vector\_game() (in module quantecon.game\_theory.game\_generators.bimatrix\_generators), 19  
 update() (quantecon.kalman.Kalman method), 56  
 update\_values() (quantecon.lqcontrol.LQ method), 59

## V

value\_iteration() (quantecon.markov.ddp.DiscreteDP method), 33  
 var (quantecon.distributions.BetaBinomial attribute), 46  
 var\_quadratic\_sum() (in module quantecon.quadsums), 73  
 vertex\_enumeration() (in module quantecon.game\_theory.vertex\_enumeration), 15

`vertex_enumeration_gen()` (in module `quantecon.game_theory.vertex_enumeration`), 15

## W

`whitener_1ss()` (`quantecon.kalman.Kalman` method), 56