# neng Documentation

***Release***

**Author**

September 19, 2013

# CONTENTS


**1 Indices and tables** **3**

**2 neng Package** **5**
- 2.1 `game` Module . . . . . 5
- 2.2 `game_reader` Module . . . . . 7
- 2.3 `strategy_profile` Module . . . . . 8
- 2.4 `cmaes` Module . . . . . 9
- 2.5 `support_enumeration` Module . . . . . 10

**Python Module Index** **13**


Neng is a tool for computing Nash equilibrium in non-cooperative games (the strategic situation where contestants/players facing each other and not making any alliances). The game is represented by payoffs or outcomes for every combination of actions of every players. Every player has at least two strategies from which he can freely choose.

Nash equilibrium is presenting to us balanced situation, thus situation where no player wants to change his strategy. It's important to tell that it does not tell us when the players has biggest outcomes.

CHAPTER

ONE

# INDICES AND TABLES

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

CHAPTER

TWO

# NENG PACKAGE

**mod** *neng* Module

---

neng.neng.**main**()
Main method of script. Based on information from arguments executes needed commands.

neng.neng.**parse_args**()
Parse arguments of script.

**Returns** parsed arguments

**Return type** dict

## 2.1 game Module

**class** neng.game.**Game**(*nfg*, *trim='normalization'*)
Bases: object

Class Game wrap around all informations of noncooperative game. Also it provides basic analyzation of game, like pureBestResponse, if the game is degenerate. It also contains an algorithm for iterative elimination of strictly dominated strategies and can compute pure Nash equilibria using brute force.

**usage:**

```
>>> g = Game(game_str)
>>> ne = g.findEquilibria(method='pne')
>>> print g.printNE(ne)
```

Initialize basic attributes in Game

**Parameters**

- **nfg** (*str*) – string containing the game in nfg format
- **trim** (*str*) – method of assuring that strategy profile lies in Delta space,'normalization'|'penalization'

**IESDS**()
Iterative elimination of strictly dominated strategies.

Eliminates all strict dominated strategies, preserve self.array and self.shape in self.init_array and self.init_shape. Stores numbers of deleted strategies in self.deleted_strategies. Deletes strategies from self.array and updates self.shape.

**LyapunovFunction**(*strategy_profile_flat*)
Lyapunov function. If LyapunovFunction(p) == 0 then p is NE.

$$x_{ij}(p) = u_i(si, p_i)$$
$$y_{ij}(p) = x_{ij}(p) - u_i(p)$$
$$z_{ij}(p) = \max[y_{ij}(p), 0]$$
$$LyapunovFunction(p) = \sum_{i \in N} \sum_{1 \leq j \leq \mu} z_{ij}(p)^2$$

Beside this function we need that strategy_profile is in universum Delta (basicaly to have character of probabilities for each player). We can assure this with two methods: normalization and penalization.

**Parameters** **strategy_profile_flat** (*list*) – list of parameters to function

**Returns** value of Lyapunov function in given strategy profile

**Return type** float

**METHODS = ['L-BFGS-B', 'SLSQP', 'CMAES', 'support_enumeration', 'pne']**

**checkBestResponses**(*strategy_profile*)
Check if every strategy of strategy profile is best response to other strategies.

**Parameters** **strategy_profile** (*StrategyProfile*) – examined strategy profile

**Returns** whether every strategy is best response to others

**Return type** bool

**checkNEs**(*nes*)
Check if given container of strategy profiles contains only Nash equlibria.

**Parameters** **nes** – container of strategy profiles to examine

**Returns** whether every strategy profile pass NE test

**Return type** boool

**findEquilibria**(*method='CMAES'*)
Find all equilibria, using method

**Parameters** **method** (*str, one of Game.METHODS*) – of computing equilibria

**Returns** list of NE, if not found returns None

**Return type** list of StrategyProfile

**getDominatedStrategies**()

**Returns** list of dominated strategies per player

**Return type** list

**getPNE**()
Function computes pure Nash equlibria using brute force algorithm.

**Returns** list of StrategyProfile that are pure Nash equilibria

**Return type** list

**isDegenerate**()
Degenerate game is defined for two-players games and there can be infinite number of mixed Nash equilibria.

**Returns** True if game is said as degenerated

**Return type** bool

**isMixedBestResponse**(*player*, *strategy_profile*)
Check if strategy of player from strategy_profile is best response for opponent strategies.

**Parameters**

- **player** (*int*) – player who should respond
- **strategy_profile** – strategy profile

**Returns** True if strategy_profile[players] is best response

**Return type** bool

**payoff**(*strategy_profile*, *player*, *pure_strategy=None*)
Function to compute payoff of given strategy_profile.

**Parameters**

- **strategy_profile** (*StrategyProfile*) – strategy profile of all players
- **player** (*int*) – player for whom the payoff is computed
- **pure_strategy** (*int*) – if not None player strategy will be replaced by pure strategy of that number

**Returns** value of payoff

**Return type** float

**printNE**(*nes*, *payoff=False*)
Print Nash equilibria with with some statistics

**Parameters**

- **nes** (*list of StrategyProfile*) – list of Nash equilibria
- **payoff** (*bool*) – flag to print payoff of each player

**Returns** string to print

**Return type** str

**pureBestResponse**(*player*, *strategy*)
Computes pure best response strategy profile for given opponent strategy and player

**Parameters**

- **player** (*int*) – player who should respond
- **strategy** (*list*) – opponnet strategy

**Returns** set of best response strategies

**Return type** set of coordinates

## 2.2 game_reader Module

**class** neng.game_reader.**GameReader**
Bases: object

Read games from different file formats (.nfg payoff, .nfg outcome), see http://www.gambit-project.org/doc/formats.html for more information.

**readFile**(*file*)
Read content of nfg file.

**Parameters** **file** (*str*) – path to file

**Returns** dictionary with game informations

**Return type** dict

**readStr**(*string*)
Base function that convert text to tokens a determine which

**Parameters** **string** (*str*) – string with nfg formated text

**Returns** dictionary with game informations

**Return type** dict

**Raise** Exception, if the string is not in specified format

neng.game_reader.**read**(*content*)

## 2.3 strategy_profile Module

**class** neng.strategy_profile.**StrategyProfile**(*profile*, *shape*, *coordinate=False*)
Bases: object

Wraps information about strategy profile of game.

**Parameters**

- **profile** (*list*) – one- level list of probability coefficients
- **shape** (*list*) – list of number of strategies per player
- **coordinate** (*bool*) – if True, then profile is considered as coordinate in game universum (depict pure strategy profile)

**copy**()
Copy constructor for StrategyProfile. Copies content of self to new object.

**Returns** StrategyProfile with same attributes

**normalize**()
Normalizes values in StrategyProfile, values can't be negative, bigger than one and sum of values of one strategy has to be 1.0.

**Returns** self

**randomize**()
Makes strategy of every player random.

**Returns** self

**randomizePlayerStrategy**(*player*)
Makes strategy of player random.

**Parameters** **player** (*int*) – player, whos strategy will be randomized

**Returns** self

**updateWithPureStrategy**(*player*, *pure_strategy*)
Replaces strategy of player with pure_strategy

**Parameters**

- **player** (*int*) – order of player
- **pure_strategy** (*int*) – order of strategy to be pure

**Returns** self

## 2.4 cmaes Module

**class** neng.cmaes.**CMAES**(*func*, *N*, *sigma=0.3*, *xmean=None*)
Bases: object

Class CMAES represent algorithm Covariance Matrix Adaptation - Evolution Strategy. It provides function minimization.

**Parameters**

- **func** (*function*) – function to be minimized
- **N** (*int*) – number of parameter of function
- **sigma** (*float*) – step size of method
- **xmean** (*np.array*) – initial point, if None some is generated

**checkStop**()
Termination criteria of method. They are checked every iteration. If any of them is true, computation should end.

**Returns** True if some termination criteria was met, False otherwise

**Return type** bool

**fmin**()
Method for actual function minimization. Iterates while not end. If unsuccess termination criteria is met then the method is restarted with doubled population. If the number of maximum evaluations is reached or the function is acceptable minimized, iterations ends and result is returned.

**Return type** scipy.optimize.Result

**initVariables**(*sigma*, *xmean*, *lamda_factor=1*)
Init variables that can change after restart of method, basically that are dependent on lamda.

**Parameters**

- **sigma** (*float*) – step size
- **xmean** (*np.array*) – initial point
- **lamda_factor** (*float*) – factor for multyplying old lambda, serves for restarting method

**logState**()
Function for logging the progress of method.

**newGeneration**()
Generate new generation of individuals.

**Return type** np.array

**Returns** new generation

**restart**(*lamda_factor*)
Restart whole method to initial state, but with population multiplied by lamda_factor.

**Parameters** **lamda_factor** (*int*) – multiply factor

**result**
Result of computation. Not returned while minimization is in progress.

**Returns** result of computation

**Return type** scipy.optimize.Result

**update**(*arfitness*)
Update values of method from new evaluated generation

**Parameters** **arfitness** (*list*) – list of function values to individuals

neng.cmaes.**fmin**(*func*, *N*)
Function for easy call function minimization from other modules.

**Parameters**

- **func** (*function*) – function to be minimized
- **N** (*int*) – number of parameters of given function

**Returns** resulting statistics of computation with result

**Return type** scipy.optimize.Result

## 2.5 support_enumeration Module

**class** neng.support_enumeration.**SupportEnumeration**(*game*)
Bases: object

Class providing support enumeration method for finding all mixed Nash equilibria in two-players games.

**getEquationSet**(*combination*, *player*, *num_supports*)
Return set of equations for given player and combination of strategies for 2 players games in support_enumeration

This function returns matrix to compute (Nisan algorithm 3.4)

For given $I$ (subset of strategies) of player 1 we can write down next equations:

$$\sum_{i \in I} x_i b_{ij} = v$$
$$\sum_{i \in I} x_i = 1$$

Where $x_i$ is probability of ith strategy, $b_{ij}$ is payoff for player 2 with strategies $i \in I, j \in J$, $v$ payoff for player 1

In matrix form (k = num_supports):

$$\begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1k} & -1 \\ b_{21} & b_{22} & \cdots & b_{2k} & -1 \\ \vdots & \vdots & \ddots & \vdots & -1 \\ b_{k1} & b_{k2} & \cdots & b_{kk} & -1 \\ 1 & 1 & \cdots & 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \\ v \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$$

Analogically for result y for player 2 with payoff matrix A

**Parameters**

- **combination** (*tuple*) – combination of strategies to make equation set

- **player** (*int*) – order of player for whom the equation will be computed
- **num_supports** (*int*) – number of supports for player

**Returns** equation matrix for solving in np.linalg.solve

**Return type** np.array

**supportEnumeration**()

Computes all mixed NE of 2 player noncooperative games. If the game is degenerate game.degenerate flag is ticked.

**Returns** list of NE computed by method support enumeration

**Return type** list

neng.support_enumeration.**computeNE**(*game*)

Function for easy calling SupportEnumeration from other modules.

**Returns** result of support enumeration algorithm

**Return type** list of StrategyProfile

# PYTHON MODULE INDEX

## n

neng.cmaes, 9
neng.game, 5
neng.game_reader, 7
neng.neng, 5
neng.strategy_profile, 8
neng.support_enumeration, 10