Nashpy Documentation

Release 0.0.23

Vincent Knight

May 04, 2021

CONTENTS

This is a Python library used for the computation of equilibria in 2 player strategic form games.

This is a library with simple dependencies (it only requires `numpy` and `scipy`) so that it is pip installable.

CHAPTER

ONE

TUTORIAL: BUILDING AND FINDING THE EQUILIBRIUM FOR A SIMPLE GAME

1.1 Introduction to game theory

Game theory is the study of strategic interactions between rational agents. Simply put that means that it's the study of interactions when the involved parties try and do what is best from their point of view.

As an example let us consider Rock Paper Scissors. This is a common game where two players choose one of 3 options (in game theory we call these *strategies*):

- Rock
- Paper
- Scissors

The winner is decided according to the following:

- Rock crushes scissors
- Paper covers Rock
- Scissors cuts paper

We can represent this mathematically using a 3 by 3 matrix:

$$A = \begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{pmatrix}$$

The matrix A_{ij} shows the utility to the player controlling the rows when they play the i th row and their opponent (the column player) plays the j th column. For example, if the row player played Scissors (the 3rd strategy) and the column player played Paper (the 2nd strategy) then the row player gets: $A_{32} = 1$ because Scissors cuts Paper.

A recommend text book on Game Theory is [Maschler2013].

1.2 Installing Nashpy

We are going to study this game using Nashpy, first though we need to install it. Nasphy requires the following things to be on your computer:

- Python 3.5 or greater;
- Scipy 0.19.0 or greater;
- Numpy 1.12.1 or greater.

Assuming you have those installed, to install Nashpy:

- On Mac OSX or linux open a terminal;
- On Windows open the Command prompt or similar

and type:

```
$ python -m pip install nashpy
```

If this does not work, you might not have Python or one of the other dependencies. You might also have problems due to `pip` not being recognised. To overcome these, using the Anaconda distribution of Python is recommended as it installs straightforwardly on all operating systems and also includes the libraries needed to run `Nashpy`.

1.3 Creating a game

We can create this game using Nashpy:

```
>>> import nashpy as nash
>>> import numpy as np
>>> A = np.array([[0, -1, 1], [1, 0, -1], [-1, 1, 0]])
>>> rps = nash.Game(A)
>>> rps
Zero sum game with payoff matrices:

Row player:
[[ 0 -1  1]
 [ 1  0 -1]
 [-1  1  0]]

Column player:
[[ 0  1 -1]
 [-1  0  1]
 [ 1 -1  0]]
```

The string representation of the game also contains some information. For example, it is also showing the matrix that corresponds to the utility of the column player. In this case that is just $-A$ but that does not always have to be the case.

We can in fact pass a pair of matrices to the game class to create the same game:

```
>>> B = - A
>>> rps = nash.Game(A, B)
>>> rps
Zero sum game with payoff matrices:

Row player:
[[ 0 -1  1]
 [ 1  0 -1]
 [-1  1  0]]

Column player:
[[ 0  1 -1]
 [-1  0  1]
 [ 1 -1  0]]
```

We get the exact same game, if passed a single game, `Nashpy` will assume that the game is a *zero sum game*: in other words the utilities of both players are opposite.

1.4 Calculating the utility of a pair of strategies

If the row player played Scissors (the 3rd strategy) and the column player played Paper (the 2nd strategy) then the row player gets: $A_{32} = 1$ because Scissors cuts Paper.

A mathematical approach to representing a strategy is to consider a vector of the size: the number of strategies. For example $\sigma_r = (0, 0, 1)$ is the row strategy where the row player always plays their third strategy. Similarly $\sigma_c = (0, 1, 0)$ is the strategy for the column player where they always play their second strategy.

When we represent strategies like this we can get the utility to the row player using the following linear algebraic expression:

$$\sigma_r A \sigma_c^T$$

Similarly, if B is the utility to the column player their utility is given by:

$$\sigma_r B \sigma_c^T$$

We can use Nashpy to find these utilities:

```
>>> sigma_r = [0, 0, 1]
>>> sigma_c = [0, 1, 0]
>>> rps[sigma_r, sigma_c]
array([ 1, -1])
```

Players can of course choose to play randomly, in which case the utility corresponds to the long term average. This is where our representation of strategies and utility calculations becomes particularly useful. For example, let us assume the column player decides to play Rock and Paper "randomly". This corresponds to $\sigma_c = (1/2, 1/2, 0)$:

```
>>> sigma_c = [1 / 2, 1 / 2, 0]
>>> rps[sigma_r, sigma_c]
array([0., 0.])
```

The row player might then decide to change their strategy and "randomly" play Paper and Scissors:

```
>>> sigma_r = [0, 1 / 2, 1 / 2]
>>> rps[sigma_r, sigma_c]
array([ 0.25, -0.25])
```

The column player would then probably deviate once more. Whether or not their is a pair of strategies for both players at which they both no longer have a reason to move is going to be answered in the next section.

1.5 Computing Nash equilibria

Nash equilibria is (in two player games) a pair of strategies at which both players do not have an incentive to deviate. We can find these using `Nashpy`:

```
>>> eqs = rps.support_enumeration()
>>> list(eqs)
[(array([0.333..., 0.333..., 0.333...]), array([0.333..., 0.333..., 0.333...]))]
```

Nash equilibria is an important concept as it allows to gain an initial understanding of emergent behaviour in complex systems.

1.6 Learning in games

Nash equilibria are not always observed during non cooperative play: they correspond to strategies at which no play has an incentive to move but that does not necessarily imply that players can arrive at that equilibria naturally.

We can illustrate this using `Nashpy`:

```
>>> import numpy as np
>>> iterations = 100
>>> np.random.seed(0)
>>> play_counts = rps.fictitious_play(iterations=iterations)
>>> for row_play_count, column_play_count in play_counts:
...     print(row_play_count, column_play_count)
[0 0 0] [0 0 0]
[1. 0. 0.] [0. 1. 0.]
...
[28. 39. 32.] [37. 26. 36.]
[29. 39. 32.] [37. 26. 37.]
```

Over time we can see the behaviour emerge, as the play counts can be normalised to give strategy vectors. Note that these will not always converge.

CHAPTER TWO

HOW TO

How to:

2.1 Install Nashpy

Nashpy currently requires Python 3.5 or above. To install from the Python Package index (PyPi) run the following command:

```
$ python -m pip install nashpy
```

To install a development version from source:

```
$ git clone https://github.com/drvinceknight/Nashpy.git
$ cd nashpy
$ python setup.py develop
```

2.2 Create a game

A game in Nashpy is created by passing 1 or 2 matrices to the nash.Game class. Here is the zero sum game matching pennies:

```
>>> import nashpy as nash
>>> import numpy as np
>>> A = np.array([[1, -1], [-1, 1]])
>>> matching_pennies = nash.Game(A)
>>> matching_pennies
Zero sum game with payoff matrices:

Row player:
[[ 1 -1]
 [-1  1]]

Column player:
[[-1  1]
 [ 1 -1]]
```

Here is the **non** zero sum game prisoners dilemma:

```
>>> import nashpy as nash
>>> import numpy as np
>>> A = np.array([[3, 0], [5, 1]])
>>> B = np.array([[3, 5], [0, 1]])
>>> prisoners_dilemma = nash.Game(A, B)
>>> prisoners_dilemma
Bi matrix game with payoff matrices:

Row player:
[[3 0]
 [5 1]]

Column player:
[[3 5]
 [0 1]]
```

2.3 Calculate utilities

A game can be passed a pair of mixed strategies (distributions over the set of pure strategies) to return the utilities. Let us create a game to illustrate this:

```
>>> import nashpy as nash
>>> import numpy as np
>>> A = np.array([[3, 0], [5, 1]])
>>> B = np.array([[3, 5], [0, 1]])
>>> prisoners_dilemma = nash.Game(A, B)
```

The utility for both players when they both play their first strategy:

```
>>> sigma_r = np.array([1, 0])
>>> sigma_c = np.array([1, 0])
>>> prisoners_dilemma[sigma_r, sigma_c]
array([3, 3])
```

The utility to both players when they play uniformly randomly across both their strategies:

```
>>> sigma_r = np.array([1 / 2, 1 / 2])
>>> sigma_c = np.array([1 / 2, 1 / 2])
>>> prisoners_dilemma[sigma_r, sigma_c]
array([2.25, 2.25])
```

2.4 Solve with support enumeration

One of the algorithms implemented in `Nashpy` is called *Support enumeration*, this is implemented as a method on the `Game` class:

```
>>> import nashpy as nash
>>> import numpy as np
>>> A = np.array([[1, -1], [-1, 1]])
>>> matching_pennies = nash.Game(A)
```

This `support_enumeration` method returns a generator of all the equilibria:

```
>>> equilibria = matching_pennies.support_enumeration()
>>> for eq in equilibria:
...     print(eq)
(array([0.5, 0.5]), array([0.5, 0.5]))
```

2.5 Solve with vertex enumeration

One of the algorithms implemented in `Nashpy` is called *Vertex enumeration*, this is implemented as a method on the `Game` class:

```
>>> import nashpy as nash
>>> import numpy as np
>>> A = np.array([[1, -1], [-1, 1]])
>>> matching_pennies = nash.Game(A)
```

This `vertex_enumeration` method returns a generator of all the equilibria:

```
>>> equilibria = matching_pennies.vertex_enumeration()
>>> for eq in equilibria:
...     print(eq)
(array([0.5, 0.5]), array([0.5, 0.5]))
```

2.6 Solve with Lemke Howson

One of the algorithms implemented in `Nashpy` is *The Lemke Howson Algorithm*. This algorithm does not return **all** equilibria and takes an input argument:

```
>>> import nashpy as nash
>>> import numpy as np
>>> A = np.array([[1, -1], [-1, 1]])
>>> matching_pennies = nash.Game(A)
>>> matching_pennies.lemke_howson(initial_dropped_label=0)
(array([0.5, 0.5]), array([0.5, 0.5]))
```

The `initial_dropped_label` is an integer between `0` and `sum(A.shape) - 1`. To iterate over all possible labels use the `lemke_howson_enumeration` which returns a generator:

```
>>> equilibria = matching_pennies.lemke_howson_enumeration()
>>> for eq in equilibria:
...     print(eq)
(array([0.5, 0.5]), array([0.5, 0.5]))
(array([0.5, 0.5]), array([0.5, 0.5]))
(array([0.5, 0.5]), array([0.5, 0.5]))
(array([0.5, 0.5]), array([0.5, 0.5]))
```

Note that this algorithm is not guaranteed to find **all** equilibria but is an efficient way of finding **an** equilibrium.

2.7 Use fictitious play

One of the learning algorithms implemented in `Nashpy` is called *Fictitious play*, this is implemented as a method on the `Game` class:

```
>>> import nashpy as nash
>>> import numpy as np
>>> A = np.array([[3, 1], [0, 2]])
>>> B = np.array([[2, 0], [1, 3]])
>>> game = nash.Game(A, B)
```

The `fictitious_play` method returns a generator of a given collection of learning steps:

```
>>> np.random.seed(0)
>>> iterations = 500
>>> play_counts = game.fictitious_play(iterations=iterations)
>>> for row_play_counts, column_play_counts in play_counts:
...     print(row_play_counts, column_play_counts)
[0 0] [0 0]
[1. 0.] [0. 1.]
...
[498.   1.] [497.   2.]
[499.   1.] [498.   2.]
```

Note that this process is stochastic:

```
>>> np.random.seed(1)
>>> play_counts = game.fictitious_play(iterations=iterations)
>>> for row_play_counts, column_play_counts in play_counts:
...     print(row_play_counts, column_play_counts)
[0 0] [0 0]
[0. 1.] [0. 1.]
...
[  0. 499.] [  0. 499.]
[  0. 500.] [  0. 500.]
```

It is also possible to pass a `play_counts` variable to give a starting point for the algorithm:

```
>>> np.random.seed(1)
>>> play_counts = (np.array([0., 500.]), np.array([0., 500.]))
>>> play_counts = game.fictitious_play(iterations=iterations, play_counts=play_counts)
>>> for row_play_counts, column_play_counts in play_counts:
...     print(row_play_counts, column_play_counts)
[  0. 500.] [  0. 500.]
[  0. 501.] [  0. 501.]
...
[  0. 999.] [  0. 999.]
[   0. 1000.] [   0. 1000.]
```

2.8 Use stochastic fictitious play

One of the learning algorithms implemented in `Nashpy` is called *Stochastic fictitious play*, this is implemented as a method on the `Game` class:

```
>>> import nashpy as nash
>>> import numpy as np
>>> A = np.array([[3, 1], [0, 2]])
>>> B = np.array([[2, 0], [1, 3]])
>>> game = nash.Game(A, B)
```

The `stochastic_fictitious_play` method returns a generator of a given collection of learning steps, comprising of the play counts and the mixed strategy of each player:

```
>>> np.random.seed(0)
>>> iterations = 500
>>> play_counts_and_distributions = game.stochastic_fictitious_
↪play(iterations=iterations)
>>> for play_counts, distributions in play_counts_and_distributions:
...     row_play_counts, column_play_counts = play_counts
...     row_distributions, column_distributions = distributions
...     print(row_play_counts, column_play_counts, row_distributions, column_
↪distributions)
[0 0] [0 0] None None
[1. 0.] [0. 1.] [9.99953841e-01 4.61594628e-05] [0.501447 0.498553]
...
[498.   1.] [497.   2.] [1.00000000e+00 1.07557011e-13] [9.99999998e-01 2.32299935e-
↪09]
[499.   1.] [498.   2.] [1.00000000e+00 1.17304491e-13] [9.99999998e-01 2.18403537e-
↪09]
```

Note that this process is stochastic:

```
>>> np.random.seed(1)
>>> play_counts_and_distributions = game.stochastic_fictitious_
↪play(iterations=iterations)
>>> for play_counts, distributions in play_counts_and_distributions:
...     row_play_counts, column_play_counts = play_counts
...     row_distributions, column_distributions = distributions
...     print(row_play_counts, column_play_counts)
[0 0] [0 0]
[1. 0.] [1. 0.]
...
[499.   0.] [499.   0.]
[500.   0.] [500.   0.]
```

It is also possible to pass a `play_counts` variable to give a starting point for the algorithm:

```
>>> np.random.seed(0)
>>> play_counts = (np.array([0., 500.]), np.array([0., 500.]))
>>> play_counts_and_distributions = game.stochastic_fictitious_
↪play(iterations=iterations, play_counts=play_counts)
>>> for play_counts, distributions in play_counts_and_distributions:
...     row_play_counts, column_play_counts = play_counts
...     row_distributions, column_distributions = distributions
...     print(row_play_counts, column_play_counts)
...
```

(continues on next page)

(continued from previous page)

```
[   0. 500.] [   0. 500.]
[   0. 501.] [   0. 501.]
...
[   0. 999.] [   0. 999.]
[    0. 1000.] [    0. 1000.]
```

A value of `etha` and `epsilon_bar` can be passed. See the *Stochastic fictitious play* reference section for more information. The default values for etha and epsilon bar are 10^-1 and 10^-2 respectively:

```
>>> np.random.seed(0)
>>> etha = 10**-2
>>> epsilon_bar = 10**-3
>>> play_counts_and_distributions = game.stochastic_fictitious_
↪play(iterations=iterations, etha=etha, epsilon_bar=epsilon_bar)
>>> for play_counts, distributions in play_counts_and_distributions:
...     row_play_counts, column_play_counts = play_counts
...     row_distributions, column_distributions = distributions
...     print(row_play_counts, column_play_counts)
...
[0 0] [0 0]
[1. 0.] [0. 1.]
...
[498.   1.] [497.   2.]
[499.   1.] [498.   2.]
```

2.9 Use replicator dynamics

One of the learning algorithms implemented in `Nashpy` is called *Replicator dynamics*, this is implemented as a method on the `Game` class:

```
>>> import nashpy as nash
>>> import numpy as np
>>> A = np.array([[3, 2], [4, 2]])
>>> game = nash.Game(A)
```

The `replicator_dynamics` method returns the strategies of the row player over time:

```
>>> game.replicator_dynamics()
array([[0.5       , 0.5       ],
       [0.49875032, 0.50124968],
       [0.49750377, 0.50249623],
...
       [0.10199196, 0.89800804],
       [0.10189853, 0.89810147],
       [0.10180527, 0.89819473]])
```

It is also possible to pass a `y0` variable in order to assign a starting strategy. Otherwise the probability is divided equally amongst all possible actions. Passing a `timepoints` variable gives the algorithm a sequence of timepoints over which to calculate the strategies:

```
>>> y0 = np.array([0.9, 0.1])
>>> timepoints = np.linspace(0, 10, 1000)
>>> game.replicator_dynamics(y0=y0, timepoints=timepoints)
array([[0.9       , 0.1       ],
```

(continues on next page)

(continued from previous page)

```
       [0.89918663, 0.10081337],
       [0.89836814, 0.10163186],
...
       [0.14109126, 0.85890874],
       [0.1409203 , 0.8590797 ],
       [0.14074972, 0.85925028]])
```

2.10 Use asymmetric replicator dynamics

This algorithm that is implemented in `Nashpy` is called *Asymmetric replicator dynamics* and is implemented as a method on the `Game` class:

```
>>> import nashpy as nash
>>> import numpy as np
>>> A = np.array([[3, 2], [4, 2]])
>>> B = np.array([[1, 3], [2, 4]])
>>> game = nash.Game(A, B)
```

The `asymmetric_replicator_dynamics` method returns the strategies of both the row player and the column player over time:

```
>>> xs, ys = game.asymmetric_replicator_dynamics()
>>> xs
array([[0.5       , 0.5       ],
       [0.49875..., 0.50124...],
       [0.49752..., 0.50247...],
       ...,
       [0.41421..., 0.58578...],
       [0.41421..., 0.58578...],
       [0.41421..., 0.58578...]])
>>> ys
array([[5.00000...e-01, 5.00000...e-01],
       [4.94995...e-01, 5.05004...e-01],
       [4.89991...e-01, 5.10008...e-01],
       ...,
       [2.28749...e-09, 9.99999...e-01],
       [2.24298...e-09, 9.99999...e-01],
       [2.19926...e-09, 9.99999...e-01]])
```

It is also possible to pass `x0` and `y0` arguments to assign the initial strategy to be played. Otherwise the probability is divided equally amongst all possible actions for both `x0` and `y0`. Additionally, a `timepoints` argument may be passed that gives the algorithm a sequence of timepoints over which to calculate the strategies.

```
>>> x0 = np.array([0.4, 0.6])
>>> y0 = np.array([0.9, 0.1])
>>> timepoints = np.linspace(0, 10, 1000)
>>> xs, ys = game.asymmetric_replicator_dynamics(x0=x0, y0=y0, timepoints=timepoints)
>>> xs
array([[0.4       , 0.6       ],
       [0.39784..., 0.60215...],
       [0.39569..., 0.60430...],
       ...,
       [0.17411..., 0.82588...],
       [0.17411..., 0.82588...],
```

(continues on next page)

(continued from previous page)

```
       [0.17411..., 0.82588...]])
>>> ys
array([[9.00000...e-01, 1.00000...e-01],
       [8.98183...e-01, 1.01816...e-01],
       [8.96338...e-01, 1.03661...e-01],
       ...,
       [1.86696...e-08, 9.99999...e-01],
       [1.82868...e-08, 9.99999...e-01],
       [1.79139...e-08, 9.99999...e-01]])
```

CHAPTER

THREE

REFERENCE

3.1 Support enumeration

The support enumeration algorithm implemented in `Nashpy` is based on the one described in [Nisan2007].

The algorithm is as follows:

For a degenerate 2 player game $(A, B) \in \mathbb{R}^{m\times n^2}$ the following algorithm returns all nash equilibria:

1. For all $1 \leq k_1 \leq m$ and $1 \leq k_2 \leq n$;
2. For all pairs of support (I, J) with $|I| = k_1$ and $|J| = k_2$.
3. Solve the following equations (this ensures we have best responses):

$$\sum_{i\in I}{\sigma_{r}}_{i}B_{ij} = v \text{ for all } j \in J$$
$$\sum_{j\in J}A_{ij}{\sigma_{c}}_{j} = u \text{ for all } i \in I$$

4. Solve
   - $\sum_{i=1}^{m}{\sigma_{r}}_{i} = 1$ and ${\sigma_{r}}_{i} \geq 0$ for all i
   - $\sum_{j=1}^{n}{\sigma_{c}}_{i} = 1$ and ${\sigma_{c}}_{j} \geq 0$ for all j
5. Check the best response condition.

Repeat steps 3,4 and 5 for all potential support pairs.

3.1.1 Discussion

1. Step 1 is a complete enumeration of all possible strategies that the equilibria could be.
2. Step 2 can be modified to only consider degenerate games ensuring that only supports of equal size are considered $|I| = |J|$. This is described further in *Degenerate games*.
3. Step 3 are the linear equations that are to be solved, for a given pair of supports these ensure that neither player has an incentive to move to another strategy on that support.
4. Step 4 is to ensure we have mixed strategies.
5. Step 5 is a final check that there is no better utility outside of the supports.

In `Nashpy` this is all implemented algebraically using `Numpy` to solve the linear equations.

3.2 Vertex enumeration

The vertex enumeration algorithm implemented in `Nashpy` is based on the one described in [Nisan2007].

The algorithm is as follows:

For a nondegenerate 2 player game $(A, B) \in \mathbb{R}^{m\times n^2}$ the following algorithm returns all nash equilibria:

1. Obtain the best response Polytopes P and Q.
2. For all pairs of vertices of P and Q.
3. Check if the pair is fully labeled and return the normalised probability vectors.

Repeat steps 2 and 3 for all pairs of vertices.

3.2.1 Discussion

1. Before creating the best response Polytope we need to consider the best response Polyhedron. For the row player, this corresponds to the set of all the mixed strategies available to the row player as well as an upper bound on the utilities to the column player. Analogously for the column player:

$$\bar{P} = \{(x, v) \in \mathbb{R}^m \times \mathbb{R} \mid x \geq 0, \nVdash x = 1, B^T x \leq \nVdash v\}$$
$$\bar{Q} = \{(y, u) \in \mathbb{R}^n \times \mathbb{R} \mid y \geq 0, \nVdash y = 1, Ay \leq \nVdash u\}$$

   Note that in both definitions above we have a total of $m + n$ inequalities in the constraints.

   For P, the first m of those constraints correspond to the elements of x being greater or equal to 0. For a given x, if $x_i = 0$, we say that x has label :math`i`. This corresponds to strategy i not being in the support of x.

   For the last n of these inequalities, when they are equalities they correspond to whether or not strategy $1 \leq j \leq n$ of the other player is a best response to x. Similarly, if strategy j is a best response to x then we say that x has label $m + j$.

   This all holds analogously for the column player. If the labels of a pair of elements of $\bar{P}$ and $\bar{Q}$ give the full set of integers from 1 to $m+n$ then they represent strategies that are best responses to each other. Since, this would imply that either a pure stragey is not played or it is a best response to the other players strategy.

   The difficulty with using the best response Polyhedron is that the upper bound on the utilities of both players (u, v) is not known. Importantly, we do not need to know it. Thus, we assume that in both cases: $u = v = 1$ (this simply corresponds to a scaling of our strategy vectors).

   This allows us to define the best response Polytopes:

$$P = \{(x, v) \in \mathbb{R}^m \times \mathbb{R} \mid x \geq 0, B^T x \leq 1\}$$
$$Q = \{(y, u) \in \mathbb{R}^n \times \mathbb{R} \mid y \geq 0, Ay \leq 1\}$$

2. Step 2: The vertices of these polytopes are the points that will have labels (they are the points that lie at the intersection of the underlying halfspaces [Ziegler2012]).

   To find these vertices, `nashpy` uses `scipy` which has a handy class for creating Polytopes using the inequality definitions and being able to return the vertices. Here is the wrapper written in `nashpy` that is used by the vertex enumeration algorithm to give the vertices and corresponding labels:

```
>>> import nashpy as nash
>>> import numpy as np
>>> A = np.array([[3, 1], [1, 3]])
>>> halfspaces = nash.polytope.build_halfspaces(A)
```

(continues on next page)

(continued from previous page)

```
>>> vertices = nash.polytope.non_trivial_vertices(halfspaces)
>>> for vertex in vertices:
...     print(vertex)
(array([0.333..., 0...]), {0, 3})
(array([0..., 0.333...]), {1, 2})
(array([0.25, 0.25]), {0, 1})
```

3. Step 3, we iterate over all pairs of the vertices of both polytopes and pick out the ones that are fully labeled. Because of the scaling that took place to create the Polytope from the Polyhedron, we will need to return a normalisation of both vertices.

3.3 The Lemke Howson Algorithm

The Lemke Howson algorithm implemented in `Nashpy` is based on the one described in [Nisan2007] originally introduced in [Lemke1964].

The algorithm is as follows:

For a nondegenerate 2 player game $(A, B) \in \mathbb{R}^{m\times n^2}$ the following algorithm returns a single Nash equilibria:

1. Obtain the best response Polytopes P and Q.
2. Choose a starting label to drop, this will correspond to a vertex of P or Q.
3. In that polytope, remove the label from the corresponding vertex and move to the vertex that shared that label. A new label will be picked up and duplicated in the other polytope.
4. In the other polytope drop the duplicate label and move to the vertex that shared that label.

Repeat steps 3 and 4 until there are no duplicate labels.

3.3.1 Discussion

This algorithm is implemented using integer pivoting.

1. Step 1, the best response polytopes P and Q are represented by a tableau. For example for:

$$A = \begin{pmatrix}3 & 1\\ 1 & 3\end{pmatrix}$$

$$B = \begin{pmatrix}1 & 3\\ 2 & 1\end{pmatrix}$$

This is represented as a pair of tableau:

$$T_c = \begin{pmatrix}3 & 1 & 1 & 0 & 1\\ 1 & 3 & 0 & 1 & 1\end{pmatrix}$$

For reasons that will become clear, we infact shift this tableau so that the labelling is coherent across both polytopes:

$$T_c = \begin{pmatrix}1 & 0 & 3 & 1 & 1\\ 0 & 1 & 1 & 3 & 1\end{pmatrix}$$

Here it is as a `numpy` array:

```
>>> import numpy as np
>>> col_tableau = np.array([[1, 0, 3, 1, 1],
...                         [0, 1, 1, 3, 1]])
```

Here is the tableau that corresponds to B:

$$T_r = \begin{pmatrix} 1 & 2 & 1 & 0 & 1 \\ 3 & 1 & 0 & 1 & 1 \end{pmatrix}$$

Here it is as a `numpy` array:

```
>>> row_tableau = np.array([[1, 2, 1, 0, 1],
...                         [3, 1, 0, 1, 1]])
```

2. Step 2, choosing a starting label is choosing an integer from $0 \leq k < m + n$ (we start our indices at 0). As an example, let us choose the label 1.

   First we need to identify which vertex has that label. The labels of a tableau correspond to the non basic variables: these are the columns with more than just a single non zero variable:

   - The labels of T_c are thus $\{2, 3\}$:

   ```
   >>> import nashpy as nash
   >>> nash.integer_pivoting.non_basic_variables(col_tableau)
   {2, 3}
   ```

   - The labels of T_r are thus $\{0, 1\}$:

   ```
   >>> nash.integer_pivoting.non_basic_variables(row_tableau)
   {0, 1}
   ```

   So we are going to drop label 1 from T_r.

3. Step 3, removing a label and moving from one vertex to another corresponds to integer pivoting [Dantzig2016]. This is a manipulation of T, dropping label 1 corresponds to pivoting the second column.

   To do this we need to identify which row will not change (the "pivot row"), this is done by finding the smallest ratio of value in that column over the value in the last column: $(T_r)_{i4}/(T_r)_{ik}$.

   In our case the first row has corresponding ratio: $1/2$ and the second ratio $1/1$. So our pivot row is the first row:

```
>>> nash.integer_pivoting.find_pivot_row(row_tableau, column_index=1)
0
```

   What we now do is row operations so as to make the second column correspond to a basic variable. We will do this by multiplying the second row by 2 and then subtracting the first row by it:

$$T_r = \begin{pmatrix} 1 & 2 & 1 & 0 & 1 \\ 5 & 0 & -1 & 2 & 1 \end{pmatrix}$$

   Our resulting tableau has labels: $\{0, 2\}$ so it has "picked up" label 2:

```
>>> nash.integer_pivoting.pivot_tableau(row_tableau, column_index=1)
{2}
>>> row_tableau
array([[ 1,  2,  1,  0,  1],
       [ 5,  0, -1,  2,  1]])
```

4. Step 4, we will now repeat the previous manipulation on T_c where we now need to drop the duplicate label 2. We do this by pivoting the third column.

   The ratios are: $1/3$ for the first row and $1/1$ for the second, thus the pivot row is the first row:

```
>>> nash.integer_pivoting.find_pivot_row(col_tableau, column_index=2)
0
```

   Using similar row operations we obtain:

$$T_c = \begin{pmatrix}1 & 0 & 3 & 1 & 1\\ -1 & 3 & 0 & 8 & 2\end{pmatrix}$$

   Our resulting tableau has labels: $\{0, 3\}$, so it has picked up label 0:

```
>>> nash.integer_pivoting.pivot_tableau(col_tableau, column_index=2)
{0}
>>> col_tableau
array([[ 1,  0,  3,  1,  1],
       [-1,  3,  0,  8,  2]])
```

   We now need to drop 0 from T_r, we do this by pivoting the first column. The ratio test: $1/1 > 1/5$ implies that the second row is the pivot row. Using similar algebraic manipulations we obtain:

$$T_r = \begin{pmatrix}0 & 10 & 6 & -2 & 4\\ 5 & 0 & -1 & 2 & 1\end{pmatrix}$$

   Our resulting tableau has labels: $\{2, 3\}$, so it has picked up label 3:

```
>>> nash.integer_pivoting.pivot_tableau(row_tableau, column_index=0)
{3}
>>> row_tableau
array([[ 0, 10,  6, -2,  4],
       [ 5,  0, -1,  2,  1]])
```

   We now need to drop 3 from T_c, we do this by pivoting the fourth column. The ratio test: $1/1 > 2/8$ indicates that we pivot on the second row which gives:

$$T_c = \begin{pmatrix}9 & -1 & 24 & 0 & 6\\ -1 & 3 & 0 & 8 & 2\end{pmatrix}$$

   Our resulting tableau has labels: $\{0, 1\}$:

```
>>> nash.integer_pivoting.pivot_tableau(col_tableau, column_index=3)
{1}
>>> col_tableau
array([[ 9, -3, 24,  0,  6],
       [-1,  3,  0,  8,  2]])
```

   The union of the labels of T_r and T_c is: $\{0, 1, 2, 3\}$ which implies that we have a fully labeled vertx pair.

   The vertex corresponding to T_r are obtained by setting the non basic variables to 0 and looking at the corresponding values of the first two columns:

$$v_r = (1/5, 4/10) = (1/5, 2/5)$$

   The vertex corresponding to T_c are obtained from the last 2 columns:

$$v_c = (6/24, 2/8) = (1/4, 1/4)$$

The final step of the algorithm is to return the normalised probabilities that correspond to these vertices:

$$((1/3, 2/3), (1/2, 1/2))$$

3.4 Degenerate games

A two player game is called nondegenerate if no mixed strategy of support size k has more than k pure best responses.

For example, the zero sum game defined by the following matrix is degenerate:

$$A = \begin{pmatrix} 0 & -1 & 1 \\ -1 & 0 & 1 \\ -1 & 1 & 0 \end{pmatrix}$$

The third column has two pure best responses.

When dealing with *degenerate* games unexpected results can occur:

```
>>> import nashpy as nash
>>> import numpy as np
>>> A = np.array([[0, -1, 1], [-1, 0, 1], [-1, 0, 1]])
>>> game = nash.Game(A)
```

Here is the output when using *Support enumeration*:

```
>>> for eq in game.support_enumeration():
...     print(np.round(eq[0], 2), np.round(eq[1], 2))
[0.5 0.5 0. ] [0.5 0.5 0. ]
[0.5 0.  0.5] [0.5 0.5 0. ]
```

Here is the output when using *Vertex enumeration*:

```
>>> for eq in game.vertex_enumeration():
...     print(np.round(eq[0], 2), np.round(eq[1], 2))
[0.5 0.  0.5] [ 0.5  0.5 -0. ]
[ 0.5  0.5 -0. ] [ 0.5  0.5 -0. ]
```

Here is the output when using the *The Lemke Howson Algorithm*:

```
>>> for eq in game.lemke_howson_enumeration():
...     print(np.round(eq[0], 2), np.round(eq[1], 2))
[0.33... 0.33... 0.33...] [nan]
```

We see that the *lemke-howson* algorithm fails but also that the *Support enumeration* and *Vertex enumeration* fail to find some equilibria: there is in fact a range of strategies the row player can play against `[ 0.5 0.5 0]` that is still a best response.

The *Support enumeration* algorithm can be executed with two optional arguments that allow for control of it's execution:

- `non_degenerate=True` (`False` is the default) will only consider supports of equal size. If you know your game is non degenerate this will make support enumeration execute less checks.
- `tol=0` (`10 ** -16` is the default), when considering the underlying linear system `tol` is considered to be a lower bound for difference between two real numbers. Using `tol=0` ensures a very strict execution of the algorithm.

Here is an example:

```
>>> A = np.array([[4, 9, 9], [9, 1, 6], [9, 2, 3]])
>>> B = np.array([[2, 2, 5], [7, 4, 4], [1, 6, 4]])
>>> game = nash.Game(A, B)
>>> for eq in game.support_enumeration():
```

(continues on next page)

(continued from previous page)

```
...     print(np.round(eq[0], 2), np.round(eq[1], 2))
[1. 0. 0.] [0. 0. 1.]
[0. 1. 0.] [1. 0. 0.]
[0.5 0.5 0. ] [0.38 0.   0.62]
[0.2 0.5 0.3] [0.57 0.32 0.11]
>>> for eq in game.support_enumeration(non_degenerate=True):
...     print(np.round(eq[0], 2), np.round(eq[1], 2))
[1. 0. 0.] [0. 0. 1.]
[0. 1. 0.] [1. 0. 0.]
[0.2 0.5 0.3] [0.57 0.32 0.11]
>>> for eq in game.support_enumeration(non_degenerate=False, tol=0):
...     print(np.round(eq[0], 2), np.round(eq[1], 2))
[1. 0. 0.] [0. 0. 1.]
[0. 1. 0.] [1. 0. 0.]
[0.2 0.5 0.3] [0.57 0.32 0.11]
```

3.5 Fictitious play

The fictitious play algorithm implemented in `Nashpy` is based on the one described in [Fudenberg1998].

The algorithm is as follows:

For a game $(A, B) \in \mathbb{R}^{m\times n}$ define $\kappa_t^i : S^{-1} \to \mathbb{N}$ to be a function that in a given time period t for a player i maps a strategy s from the opponent's strategy space S^{-1} to a number of total times the opponent has played s.

Thus:

$$\kappa_t^i(s^{-i}) = \kappa_{t-1}(s^{-i}) + \begin{cases} 1, & \text{if } s_{t-1}^{-i} = s^{-i} \\ 0, & \text{otherwise} \end{cases}$$

In practice:

$$\kappa_t^1 \in \mathbb{Z}^n \qquad \kappa_t^2 \in \mathbb{Z}^m$$

At stage t, each player assumes their opponent is playing a mixed strategy based on κ_{t-1}:

$$\frac{\kappa_{t-1}}{\sum \kappa_{t-1}}$$

They calculate the expected value of each strategy, which is equivalent to:

$$s_t^1 \in \text{argmax}_{s\in S_1} A\kappa_{t-1}^2 \qquad s_t^2 \in \text{argmax}_{s\in S_2} B^T\kappa_{t-1}^1$$

In the case of multiple best responses, a random choice is made.

3.5.1 Discussion

Note that this algorithm will not always converge and sometimes it depends on the form of the game.

For example:

```
>>> import numpy as np
>>> import nashpy as nash
>>> A = np.array([[0, 1, 0], [0, 0, 1], [1, 0, 0]])
>>> B = np.array([[0, 0, 1], [1, 0, 0], [0, 1, 0]])
>>> game = nash.Game(A, B)
>>> iterations = 10000
>>> np.random.seed(0)
>>> play_counts = tuple(game.fictitious_play(iterations=iterations))
>>> play_counts[-1]
[array([5464., 1436., 3100.]), array([2111., 4550., 3339.])]
```

We can visualise the lack of convergence:

```
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> probabilities = [row_play_counts / np.sum(row_play_counts) for row_play_counts,␣
↪col_play_counts in play_counts]
>>> for number, strategy in enumerate(zip(*probabilities)):
...     plt.plot(strategy, label=f"$s_{number}$")
>>> plt.xlabel("Iteration")
>>> plt.ylabel("Probability")
>>> plt.title("Actions taken by row player")
>>> plt.legend()
```

If we modify the game slightly we obtain a different outcome:

```
>>> A = np.array([[1 / 2, 1, 0], [0, 1 / 2, 1], [1, 0, 1 / 2]])
>>> B = np.array([[1 / 2, 0, 1], [1, 1 / 2, 0], [0, 1, 1 / 2]])
>>> game = nash.Game(A, B)
>>> np.random.seed(0)
>>> play_counts = tuple(game.fictitious_play(iterations=iterations))
>>> play_counts[-1]
[array([3290., 3320., 3390.]), array([3356., 3361., 3283.])]
```

With a clear convergence now visible:

```
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> probabilities = [row_play_counts / np.sum(row_play_counts) for row_play_counts,␣
↪col_play_counts in play_counts]
>>> for number, strategy in enumerate(zip(*probabilities)):
...     plt.plot(strategy, label=f"$s_{number}$")
>>> plt.xlabel("Iteration")
>>> plt.ylabel("Probability")
>>> plt.title("Actions taken by row player")
>>> plt.legend()
```

3.6 Stochastic fictitious play

The stochastic fictitious play algorithm implemented in `Nashpy` is based on the one given in [Hofbauer2002].

As explained in [Fudenberg1998] stochastic fictitious play "avoids the discontinuity inherent in standard fictitious play, where a small change in the data can lead to an abrupt change in behaviour."

The algorithm is designed to converge in cases where fictitious play does not converge. Note that in some cases this will require a thoughtful choice of the `etha` and `epsilon_bar` parameters.

For a game $(A, B) \in \mathbb{R}^{m\times n}$ define $\kappa_t^i : S^{-1} \to \mathbb{N}$ to be a function that in a given time period t for a player i maps a strategy s from the opponent's strategy space S^{-1} to a number of total times the opponent has played s.

As per standard *Fictitious play*, each player assumes their opponent is playing a mixed strategy based on κ_{t-1}. If no play has taken place, then the probability of playing each action is assumed to be equal. The assumed mixed strategies of a player's opponent are multplied by the player's own payoff matrices to calculate the expected payoff of each action.

A stochastic pertubation ϵ_i is added to each expected payoff π_i to give a pertubated payoff. Each ϵ_i is independent of each π_i and is a random number between 0 and `epsilon_bar`.

A logit choice function is used to map the pertubated payoff to a non-negative probability distribution, corresponding to the probability with which each strategy is chosen by the player. The logit choice function can be seen below:

$$L_i(\pi) = \frac{\exp(\eta^{-1}\pi_i)}{\sum_j \exp(\eta^{-1}\pi_j)}$$

3.6.1 Discussion

Using the same game from the fictitious play discussion section, we can visualise a lack of convergence when using the default value of `epsilon_bar`:

```
>>> import numpy as np
>>> import nashpy as nash
>>> A = np.array([[0, 1, 0], [0, 0, 1], [1, 0, 0]])
>>> B = np.array([[0, 0, 1], [1, 0, 0], [0, 1, 0]])
>>> game = nash.Game(A, B)
>>> iterations = 10000
>>> np.random.seed(0)
>>> play_counts_and_distribuions = tuple(game.stochastic_fictitious_
↪play(iterations=iterations))
>>> play_counts, distributions = play_counts_and_distribuions[-1]
>>> print(play_counts)
[array([3937., 1907., 4156.]), array([2823., 5458., 1719.])]

>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> probabilities = [
...     row_play_counts / np.sum(row_play_counts)
...     if np.sum(row_play_counts) != 0
...     else row_play_counts + 1 / len(row_play_counts)
...     for (row_play_counts, col_play_counts), _ in play_counts_and_distribuions]
>>> for number, strategy in enumerate(zip(*probabilities)):
...     plt.plot(strategy, label=f"$s_{number}$")
>>> plt.xlabel("Iteration")
>>> plt.ylabel("Probability")
>>> plt.title("Actions taken by row player")
>>> plt.legend()
```

Observe below that the game converges when passing values for `etha` and `epsilon_bar`:

```
>>> A = np.array([[1 / 2, 1, 0], [0, 1 / 2, 1], [1, 0, 1 / 2]])
>>> B = np.array([[1 / 2, 0, 1], [1, 1 / 2, 0], [0, 1, 1 / 2]])
>>> game = nash.Game(A, B)
>>> iterations = 10000
>>> etha = 0.1
>>> epsilon_bar = 10**-1
>>> np.random.seed(0)
>>> play_counts_and_distribuions = tuple(game.stochastic_fictitious_
↪play(iterations=iterations, etha=etha, epsilon_bar=epsilon_bar))
>>> play_counts_and_distribuions[-1]
([array([3300., 3293., 3407.]), array([3320., 3372., 3308.])], [array([0.33502382, 0.
↪41533594, 0.24964024]), array([0.18890743, 0.42793694, 0.38315563])])
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> probabilities = [
...     row_play_counts / np.sum(row_play_counts)
...     if np.sum(row_play_counts) != 0
...     else row_play_counts + 1 / len(row_play_counts)
...     for (row_play_counts, col_play_counts), _ in play_counts_and_distribuions]
>>> for number, strategy in enumerate(zip(*probabilities)):
...     plt.plot(strategy, label=f"$s_{number}$")
>>> plt.xlabel("Iteration")
>>> plt.ylabel("Probability")
>>> plt.title("Actions taken by row player")
>>> plt.legend()
```

3.7 Replicator dynamics

The replicator dynamic algorithm implemented in `Nashpy` is based on the one described in [Fudenberg1998].

Strategies are assigned amongst the popoulation. Individuals randomly encounter other individuals and play their assigned strategy.

As the game continues, the proportion of the population playing each strategy increases or decreases depending on whether the payoff of the strategy is higher or lower respectively than the mean payoff of the population.

The row player represents a given individual and the column player is the population.

Given a matrix $A \in \mathbb{R}^{m \times n}$ that corresponds to the utilities of the row player, we have:

$$f = Ax$$

Where $f \in \mathbb{R}^{m \times 1}$ corresponds to the fitness of each strategy and $x \in \mathbb{R}^{m \times 1}$ corresponds to the population size of each strategy

Equivalently, where ϕ equals the average fitness of the population, we have:

$$\phi = fx$$

In matrix formation we can calculate the rate of change of the strategies:

$$\frac{dx}{dt}_i = x_i(f_i - \phi) \text{ for all } i$$

3.7.1 Discussion

Stability is acheived in replicator dynamics when $\frac{dx}{dt} = 0$. Every stable steady state is a Nash equilibria, and every Nash equilibria is a steady state in replicator dynamics.

A steady state is when the population shares of all strategies are constant.

Steady states are reached when either:

- An entire population plays the same strategy
- A population plays a mixture of the strategies (such that there is indifference between the fitness)

It is possible that the game does not converge to a steady state. See below an example of a game of Rock, Paper, Scissors that does not converge:

```
>>> import numpy as np
>>> import nashpy as nash
>>> import matplotlib.pyplot as plt
>>> A = np.array([[0, -1, 1], [1, 0, -1], [-1, 1, 0]])
>>> game = nash.Game(A)
>>> y0 = np.array([0.3, 0.35, 0.35])
```

```
>>> plt.plot(game.replicator_dynamics(y0=y0))
>>> plt.xlabel("Timepoints")
>>> plt.ylabel("Probability")
>>> plt.title("Probability distribution of strategies over time")
>>> plt.legend([f"$s_{0}$", f"$s_{1}$", f"$s_{2}$"], loc='upper left')
```

Below shows an example of a stable steady state:

```
>>> import numpy as np
>>> import nashpy as nash
>>> import matplotlib.pyplot as plt
>>> A = np.array([[4, 3], [2, 3]])
>>> game = nash.Game(A)
>>> y0 = np.array([1 / 2, 1 / 2])
>>> timepoints = np.linspace(0, 10, 1000)
```

```
>>> plt.plot(game.replicator_dynamics(y0=y0, timepoints=timepoints))
>>> plt.xlabel("Timepoints")
>>> plt.ylabel("Probability")
>>> plt.title("Probability distribution of strategies over time")
>>> plt.legend([f"$s_{0}$", f"$s_{1}$"])
```

Evolutionary stable strategies (ESS) remain stable subject to small evolutionary change. This means that the strategy cannot be invaded by any of the other strategies in the population. Every ESS is an asymptotically stable steady state of the replicator dynamic, but the converse does not necessarily hold.

To visualise an example of ESS consider the matrix $A = \begin{pmatrix} 4 & 3 \\ 2 & 3 \end{pmatrix}$. It can be shown that $(1, 0)$ is an ESS for this game. Below we take a small change from this strategy and note that the replicator dynamics guide us back to it.

```
>>> import numpy as np
>>> import nashpy as nash
```

(continues on next page)

(continued from previous page)

```
>>> import matplotlib.pyplot as plt
>>> A = np.array([[4, 3], [2, 3]])
>>> game=nash.Game(A)
>>> epsilon = 1 / 10
>>> y0 = np.array([1 - epsilon, 0 + epsilon])
>>> timepoints = np.linspace(0, 10, 1000)
>>> timepoints[-1]
10.0
```

```
>>> plt.plot(game.replicator_dynamics(y0=y0, timepoints=timepoints))
>>> plt.xlabel("Timepoints")
>>> plt.ylabel("Probability")
>>> plt.title("Probability distribution of strategies over time")
>>> plt.legend([f"$s_{0}$", f"$s_{1}$"])
```

3.8 Asymmetric replicator dynamics

The asymmetric replicator dynamics algorithm is implemented in `Nashpy` based on the work presented in [Elvio2011]. This is considered as the asymmetric version of the symmetric *Replicator dynamics*.

There exists a population with two types of individuals where each type has their own strategy set. Strategies are assigned amongst the population. Individuals randomly encounter individuals of the opposite type and play their assigned strategies.

As the game progresses the proportion of each type playing each strategy changes based on their previous interactions.

The row player represents the first type of individuals and the column player represents the other one.

Given two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{m \times n}$ that correspond to the utilities of the row player and column player respectively, we define:

$$f_x = Ay$$
$$f_y = x^T B$$

Where $x \in \mathbb{R}^{m \times 1}$ and $y \in \mathbb{R}^{n \times 1}$ corresponds to the population size of the strategies of the two players and $f_x \in \mathbb{R}^{n \times 1}$ and $f_y \in \mathbb{R}^{1 \times m}$ corresponds to the fitness of the strategies of the row player and the column player respectively.

Similarly, the average fitness for the two types of populations is given by ϕ_x and ϕ_y where:

$$\phi_x = f_x x^T$$
$$\phi_y = f_y y$$

In matrix notation the rate of change of the strategies of both types of individuals is captured by:

$$\frac{dx}{dt}_i = x_i((f_x)_i - \phi_x) \text{ for all } i$$
$$\frac{dy}{dt}_i = y_i((f_y)_i - \phi_y) \text{ for all } i$$

3.8.1 Discussion

Stability is achieved in asymmetric replicator dynamics when both $\frac{dx}{dt} = 0$ and $\frac{dy}{dt} = 0$. Every stable steady state is a Nash equilibria, and every Nash equilibria is a steady state in asymmetric replicator dynamics.

Similarly to *Replicator dynamics*, a game is not guaranteed to converge to a steady state. Find below the probability distributions for both the row player and the column player over time, of a game that does not converge:

```
>>> import matplotlib.pyplot as plt
>>> import nashpy as nash
>>> import numpy as np
>>> A = np.array([[0, -1, 1], [1, 0, -1], [-1, 1, 0]])
>>> B = A.transpose()
>>> game = nash.Game(A, B)
>>> x0 = np.array([0.3, 0.35, 0.35])
>>> y0 = np.array([0.3, 0.35, 0.35])
>>> xs, ys = game.asymmetric_replicator_dynamics(x0=x0, y0=y0)
```

```
>>> plt.figure(figsize=(15, 5))
>>> plt.subplot(1, 2, 1)
>>> plt.plot(xs)
>>> plt.title("Probability distribution of strategies over time for row player")
>>> plt.legend([f"$s_{0}$", f"$s_{1}$", f"$s_{2}$"])
>>> plt.subplot(1, 2, 2)
>>> plt.plot(ys)
>>> plt.xlabel("Timepoints")
>>> plt.ylabel("Probability")
>>> plt.title("Probability distribution of strategies over time for column player")
>>> plt.legend([f"$s_{0}$", f"$s_{1}$", f"$s_{2}$"])
```

Find below an example of a game that is able to reach a stable steady state:

```
>>> import matplotlib.pyplot as plt
>>> import nashpy as nash
>>> import numpy as np
>>> A = np.array([[2, 2], [3, 4]])
>>> B = np.array([[4, 3], [3, 2]])
>>> game = nash.Game(A, B)
>>> x0 = np.array([0.9, 0.1])
>>> y0 = np.array([0.3, 0.7])
>>> xs, ys = game.asymmetric_replicator_dynamics(x0=x0, y0=y0)
```

```
>>> plt.figure(figsize=(15, 5))
>>> plt.subplot(1, 2, 1)
>>> plt.plot(xs)
>>> plt.title("Probability distribution of strategies over time for row player")
>>> plt.legend([f"$s_{0}$", f"$s_{1}$"])
>>> plt.subplot(1, 2, 2)
>>> plt.plot(ys)
>>> plt.xlabel("Timepoints")
>>> plt.ylabel("Probability")
>>> plt.title("Probability distribution of strategies over time for column player")
>>> plt.legend([f"$s_{0}$", f"$s_{1}$"])
```

3.9 Bibliography

This is a collection of various bibliographic items referenced in the documentation.

3.10 Source files

3.10.1 Subpackages

nash.algorithms package

Submodules

nashpy.algorithms.support_enumeration module

A class for a normal form game

`nashpy.algorithms.support_enumeration.`**`indifference_strategies`**(*A*, *B*, *non_degenerate=False*, *tol=1e-16*)

A generator for the strategies corresponding to the potential supports

Returns

- *A generator of all potential strategies that are indifferent on each*
- *potential support. Return False if they are not valid (not a*
- *probability vector OR not fully on the given support).*

`nashpy.algorithms.support_enumeration.`**`is_ne`**(*strategy_pair*, *support_pair*, *payoff_matrices*)

Test if a given strategy pair is a pair of best responses

Parameters

- **`strategy_pair`** (*`a 2-tuple of numpy arrays`*) –
- **`support_pair`** (*`a 2-tuple of numpy arrays`*) –

`nashpy.algorithms.support_enumeration.`**`obey_support`**(*strategy*, *support*, *tol=1e-16*)

Test if a strategy obeys its support

Parameters

- **`strategy`** (*`a numpy array`*) – A given strategy vector
- **`support`** (*`a numpy array`*) – A strategy support

Returns

- **A boolean** (*whether or not that strategy does indeed have the given*)
- *support*

`nashpy.algorithms.support_enumeration.`**`potential_support_pairs`**(*A*, *B*, *non_degenerate=False*)

A generator for the potential support pairs

Returns

Return type A generator of all potential support pairs `nashpy.algorithms.support_enumeration.`**`powerset`**(*n*)

A power set of range(n)

Based on recipe from python itertools documentation:

https://docs.python.org/2/library/itertools.html#recipes

`nashpy.algorithms.support_enumeration.`**`solve_indifference`**(*A*, *rows=None*, *columns=None*)

Solve the indifference for a payoff matrix assuming support for the strategies given by columns

Finds vector of probabilities that makes player indifferent between rows. (So finds probability vector for corresponding column player)

Parameters

- **A** (*a 2 dimensional numpy array (A payoff matrix for the row player)*) –
- **rows** (*the support played by the row player*) –
- **columns** (*the support player by the column player*) –

Returns

- *A numpy array*
- *A probability vector for the column player that makes the row*
- *player indifferent. Will return False if all entries are not >= 0.*

`nashpy.algorithms.support_enumeration.`**`support_enumeration`**(*A*, *B*, *non_degenerate=False*, *tol=1e-16*)

Obtain the Nash equilibria using support enumeration.

Algorithm implemented here is Algorithm 3.4 of [Nisan2007]

1. For each k in 1... min(size of strategy sets)
2. For each I,J supports of size k
3. Solve indifference conditions
4. Check that have Nash Equilibrium.

Returns equilibria

Return type A generator.

nashpy.algorithms.vertex_enumeration module

A class for the vertex enumeration algorithm

`nashpy.algorithms.vertex_enumeration.`**`vertex_enumeration`**(*A*, *B*)

Obtain the Nash equilibria using enumeration of the vertices of the best response polytopes.

Algorithm implemented here is Algorithm 3.5 of [Nisan2007]

1. Build best responses polytopes of both players
2. For each vertex pair of both polytopes
3. Check if pair is fully labelled
4. Return the normalised pair

Returns equilibria

Return type A generator.

nashpy.algorithms.lemke_howson module

A class for the Lemke Howson algorithm

`nashpy.algorithms.lemke_howson.`**`lemke_howson`**`(`*A*, *B*, *initial_dropped_label=0*`)`

Obtain the Nash equilibria using the Lemke Howson algorithm implemented using integer pivoting.

Algorithm implemented here is Algorithm 3.6 of [Nisan2007].

1. Start at the artificial equilibrium (which is fully labeled)
2. Choose an initial label to drop and move in the polytope for which the vertex has that label to the edge that does not share that label. (This is implemented using integer pivoting)
3. A label will now be duplicated in the other polytope, drop it in a similar way.
4. Repeat steps 2 and 3 until have Nash Equilibrium.

Parameters **`initial_dropped_label`** (*`int`*) –

Returns equilibria

Return type A tuple.

`nashpy.algorithms.lemke_howson.`**`shift_tableau`**`(`*tableau*, *shape*`)`

Shift a tableau to ensure labels of pairs of tableaux coincide

Parameters

- **`tableau`** (*`a numpy array`*) –
- **`shape`** (*`a tuple`*) –

Returns tableau

Return type a numpy array

`nashpy.algorithms.lemke_howson.`**`tableau_to_strategy`**`(`*tableau*, *basic_labels*, *strategy_labels*`)`

Return a strategy vector from a tableau

Parameters

- **`tableau`** (*`a numpy array`*) –
- **`basic_labels`** (*`a set`*) –
- **`strategy_labels`** (*`a set`*) –

Returns strategy

Return type a numpy array

nash.learning package

Submodules

nashpy.learning.fictitious_play module

Code to carry out fictitious learning

`nashpy.learning.fictitious_play.`**`fictitious_play`**(*A*, *B*, *iterations*, *play_counts=None*)
  Implement fictitious play

`nashpy.learning.fictitious_play.`**`get_best_response_to_play_count`**(*A*, *play_count*)
  Returns the best response to a belief based on the playing distribution of the opponent

`nashpy.learning.fictitious_play.`**`update_play_count`**(*play_count*, *play*)
  Update a belief vector with a given play

3.10.2 Submodules

3.10.3 nashpy.game module

A class for a normal form game

`class` `nashpy.game.`**`Game`**(**args*)
  Bases: `object`

  A class for a normal form game.

  Parameters

  - **`A`** (*–*) – non zero sum games.
  - **`B`** (*`2 dimensional list/arrays representing the payoff matrices for`*) – non zero sum games.
  - **`A`** – zero sum game.

`asymmetric_replicator_dynamics`(*x0=None*, *y0=None*, *timepoints=None*)
  Returns two arrays, corresponding to the two players, showing the probability of each strategy being played over time using the asymmetric replicator dynamics algorithm.

  Parameters

  - **`x0`** (*`array, optional`*) –
  - **`y0`** (*`array, optional`*) –
  - **`timepoints`** (*`array, optional`*) –

  Returns

  - **xs1** (*array*)
  - **xs2** (*array*)

`fictitious_play`(*iterations*, *play_counts=None*)
  Return a given sequence of actions through fictitious play. The implementation corresponds to the description of chapter 2 of [Fudenberg1998].

1. Players have a belief of the strategy of the other player: a vector representing the number of times the player has chosen a given strategy. 2. Players choose a best response to the belief. 3. Players update their belief based on the latest choice of the opponent.

Parameters

- **`iterations`** (*int*) –
- **`play_counts`** (*iterator*) –

Returns plays

Return type A generator

`lemke_howson`(*initial_dropped_label*)

Obtain the Nash equilibria using the Lemke Howson algorithm implemented using integer pivoting.

Algorithm implemented here is Algorithm 3.6 of [Nisan2007].

1. Start at the artificial equilibrium (which is fully labeled)
2. Choose an initial label to drop and move in the polytope for which the vertex has that label to the edge that does not share that label. (This is implemented using integer pivoting)
3. A label will now be duplicated in the other polytope, drop it in a similar way.
4. Repeat steps 2 and 3 until have Nash Equilibrium.

Parameters **`initial_dropped_label`** (*int*) –

Returns equilibria

Return type A tuple.

`lemke_howson_enumeration`()

Obtain Nash equilibria for all possible starting dropped labels using the lemke howson algorithm. See *Game.lemke_howson* for more information.

Note: this is not guaranteed to find all equilibria.

Returns equilibria

Return type A generator

`replicator_dynamics`(*y0=None*, *timepoints=None*)

Implement replicator dynamics Return an array showing probability of each strategy being played over time. The total population is constant. Strategies can either stay constant if equilibria is achieved, replicate or die.

Parameters

- **`A`** (*nxm array, where n=m*) –
- **`y0`** (*array*) –
- **`timepoints`** (*array*) –

Returns xs

Return type array

`stochastic_fictitious_play`(*iterations*, *play_counts=None*, *etha=0.1*, *epsilon_bar=0.01*)

Return a given sequence of actions and mixed strategies through stochastic fictitious play. The implementation corresponds to the description given in [Hofbauer2002].

Parameters

- **iterations** (*int*) –
- **play_counts** (*iterator*) –
- **etha** (*float*) –
- **epsilon_bar** (*float*) –

Returns **plays**

Return type A generator

support_enumeration(*non_degenerate=False*, *tol=1e-16*)

Obtain the Nash equilibria using support enumeration.

Algorithm implemented here is Algorithm 3.4 of [Nisan2007].

1. For each k in 1. . . min(size of strategy sets)
2. For each I,J supports of size k
3. Solve indifference conditions
4. Check that have Nash Equilibrium.

Returns **equilibria**

Return type A generator.

vertex_enumeration()

Obtain the Nash equilibria using enumeration of the vertices of the best response polytopes.

Algorithm implemented here is Algorithm 3.5 of [Nisan2007].

1. Build best responses polytopes of both players
2. For each vertex pair of both polytopes
3. Check if pair is fully labelled
4. Return the normalised pair

Returns **equilibria**

Return type A generator.

3.10.4 Module contents

A library to compute equilibria of 2 player normal form games

CHAPTER

FOUR

DISCUSSION

4.1 John Nash

This library is named after the mathematician John Nash. He is most famous for his work in Game Theory that culminated in him winning a Noble prize in Economics. The book [Nasar2011] (popularized in a 2001 movie) gives a good overview of his life.

The work he received a Noble prize for was a proof that a game **always** has an equilibrium [Nash1950]. His proof is an exceptional piece of mathematics where he uses a fixed point theorem by showing that an equilibrium is equivalent to a fixed point of a function.

Subsequently, these equilibria have been referred to as Nash equilibria.

4.2 How does Nashpy relate to Gambit

Gambit is the state of the art software library for Game Theory [McKelvey2016]. It also has a Python interface. It handles $N \geq 2$ player games and is computationally efficient. It is a much more mature piece of software than `Nashpy`.

It does **however** sometimes prove difficult to install (because of the required C libraries), in particular installation is not supported on Windows. In those instances you can use Game Theory Explorer which is a great web point and click Graphical User Interface (GUI) to Gambit.

The main mission statement of `Nashpy` is to provide a simple to install Python library that implements algorithms that are simple to implement using the scientific Python stack (`numpy` and `scipy`).

This is motivated by the fact that I wanted a Python library (not a GUI as I am keen to teach reproducibly research methodologies) for teaching my Mathematics students. Using the Gambit Python interface is not sufficient for this as students need to be able to install it on their own machines (without difficulty).

All the algorithms in `Nashpy` are implemented with readability as the main motivation. This at times comes at an efficiency cost. For example, *Support enumeration* builds the entire Polytope representation (using functionality of `scipy`) which is not efficient.

To summarise:

- If you want to do sophisticated efficient game theoretic computations, use Gambit.
- If you are happy to use a GUI use Game Theory Explorer.
- If you would like an easy to install Python library for two player games you can use `Nashpy`.

4.3 Other Python Game theory libraries

- Axelrod: a research library aimed at the study of the Iterated Prisoners dilemma [Knight2016].
- Gambit: a C library with a Python interface for the computation of equilibria [McKelvey2016]. *How does Nashpy relate to Gambit*.
- Game theory explorer a web interface to gambit useful for teaching. [Savani2015]
- PyNFG: PyNFG is a Python package for modeling and solving Network Form Games.
- lrslib: A C implementation of a reverse search algorithm with modules for Nash equilibria computation.
- sagemath: The mathematical software package Sage has various algorithms for the computation of Nash equilibria.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Dantzig2016] Dantzig, George. Linear programming and extensions. Princeton university press, 2016. APA

[Elvio2011] Elvio, Accinelli and Carrera, Edgar. 2011. Evolutionarily Stable Strategies and Replicator Dynamics in Asymmetric Two-Population Games. 10.1007/978-3-642-11456-4_3.

[Fudenberg1998] Fudenberg, Drew, et al. The theory of learning in games. Vol. 2. MIT press, 1998.

[Hofbauer2002] Hofbauer, Josef, and William H. Sandholm. “On the global convergence of stochastic fictitious play.” Econometrica 70.6 (2002): 2265-2294.

[Knight2016] Knight, V. et al., (2016). An Open Framework for the Reproducible Study of the Iterated Prisoner’s Dilemma. Journal of Open Research Software. 4(1), p.e35. DOI: http://doi.org/10.5334/jors.125

[Lemke1964] Lemke, Carlton E., and Joseph T. Howson, Jr. “Equilibrium points of bimatrix games.” Journal of the Society for Industrial and Applied Mathematics 12.2 (1964): 413-423.

[Maschler2013] Maschler, M., Eilon Solan, and Shmuel Zamir. “Game theory. Translated from the Hebrew by Ziv Hellman and edited by Mike Borns.” (2013).

[McKelvey2016] McKelvey, Richard D., McLennan, Andrew M., and Turocy, Theodore L. (2016). Gambit: Software Tools for Game Theory, Version 16.0.1. http://www.gambit-project.org.

[Nasar2011] Nasar, Sylvia. A beautiful mind. Simon and Schuster, 2011. APA

[Nash1950] Nash, John F. “Equilibrium points in n-person games.” Proceedings of the national academy of sciences 36.1 (1950): 48-49.

[Nisan2007] Nisan, Noam, et al., eds. Algorithmic game theory. Vol. 1. Cambridge: Cambridge University Press, 2007.

[Savani2015] Rahul Savani and Bernhard von Stengel. Game Theory Explorer – Software for the Applied Game Theorist. Computational Management Science 12, 5-33, 2015

[Ziegler2012] Ziegler, Günter M. Lectures on polytopes. Vol. 152. Springer Science & Business Media, 2012. APA

PYTHON MODULE INDEX

INDEX