Nashpy Documentation

Release 0.0.35

Vincent Knight

Jun 20, 2022

CONTENTS

1 Tutorial: building and finding the equilibrium for a game **3**
- 1.1 Introduction to game theory . . . 3
- 1.2 Installing Nashpy . . . 3
- 1.3 Creating a game . . . 4
- 1.4 Calculating the utility of a pair of strategies . . . 5
- 1.5 Computing Nash equilibria . . . 5
- 1.6 Learning in games . . . 6

2 How to **7**
- 2.1 Install Nashpy . . . 7
- 2.2 Create a Normal Form Game . . . 7
- 2.3 Calculate utilities . . . 8
- 2.4 Check if a strategy is a best response . . . 8
- 2.5 Solve with support enumeration . . . 9
- 2.6 Solve with vertex enumeration . . . 9
- 2.7 Solve with Lemke Howson . . . 10
- 2.8 Use fictitious play . . . 10
- 2.9 Use stochastic fictitious play . . . 11
- 2.10 Use replicator dynamics . . . 13
- 2.11 Use replicator dynamics with mutation . . . 13
- 2.12 Use asymmetric replicator dynamics . . . 14
- 2.13 Use Moran processes . . . 15
- 2.14 Obtain fixation probabilities . . . 16
- 2.15 Obtain a repeated game . . . 16

3 Discussion **19**
- 3.1 Normal Form Games . . . 19
- 3.2 Strategies . . . 24
- 3.3 Best responses . . . 27
- 3.4 Support enumeration . . . 32
- 3.5 Vertex enumeration . . . 36
- 3.6 Extensive Form Games . . . 37
- 3.7 Repeated Games . . . 46
- 3.8 The Emergence of Cooperation . . . 50
- 3.9 The Lemke Howson Algorithm . . . 54
- 3.10 Degenerate games . . . 57
- 3.11 Fictitious play . . . 58
- 3.12 Stochastic fictitious play . . . 60
- 3.13 Replicator dynamics . . . 62
- 3.14 Asymmetric replicator dynamics . . . 72

3.15 Moran Processes . . . 74

4 Reference **83**
4.1 John Nash . . . 83
4.2 How does Nashpy relate to Gambit . . . 83
4.3 Other Python Game theory libraries . . . 84
4.4 Bibliography . . . 84
4.5 Source files . . . 84

5 Contributor documentation **95**
5.1 Tutorial: make a contribution to the documentation . . . 95
5.2 How to . . . 102
5.3 Discussion . . . 110
5.4 Reference . . . 127

6 Indices and tables **129**

7 Indices and tables **131**

Bibliography **133**

Python Module Index **135**

Index **137**

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

CHAPTER

ONE

TUTORIAL: BUILDING AND FINDING THE EQUILIBRIUM FOR A GAME

1.1 Introduction to game theory

Game theory is the study of strategic interactions between rational agents. This means that it is 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 $-A$ 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 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 -m pip install flit
$ python -m flit install --symlink
```

2.2 Create a Normal Form 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 *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 action:

```
>>> 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 actions:

```
>>> 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 Check if a strategy is a best response

A game can be passed a pair of *Strategies* to check if they are best responses to each other. 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 `is_best_response` method returns a pair of booleans. In this instance, the row player strategy is a best response to the column player's but not vice versa:

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

2.5 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]))
```

2.6 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]))
```

2.7 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.

2.8 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.]
```

2.9 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)
...
[  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.]
```

2.10 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       ],
       [0.89918663, 0.10081337],
       [0.89836814, 0.10163186],
...
       [0.14109126, 0.85890874],
       [0.1409203 , 0.8590797 ],
       [0.14074972, 0.85925028]])
```

2.11 Use replicator dynamics with mutation

Given a matrix Q such that Q_{ij} gives the probability that an individual of type i mutates to an individual of type j, the replicator dynamics equation with mutation can be solved using the following:

```
>>> import nashpy as nash
>>> import numpy as np
>>> A = np.array([[3, 2], [4, 2]])
>>> Q = np.array([[9 / 10,  1 / 10], [1 / 5, 4 / 5]])
>>> game = nash.Game(A)
>>> game.replicator_dynamics(mutation_matrix=Q)
array([[0.5       , 0.5       ],
       [0.50049813, 0.49950187],
       [0.50099155, 0.49900845],
       ...,
```

(continues on next page)

(continued from previous page)

```
       [0.55278206, 0.44721794],
       [0.5527821 , 0.4472179 ],
       [0.55278214, 0.44721786]])
```

2.12 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...],
```

(continues on next page)

(continued from previous page)

```
       [0.17411..., 0.82588...],
       [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]])
```

2.13 Use Moran processes

Moran processes are implemented in `Nashpy` as a method on the `Game` class:

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

The `moran_process` method returns a generator of a given collection of generations:

```
>>> np.random.seed(0)
>>> generations = game.moran_process(initial_population=(0, 0, 1))
>>> for population in generations:
...     print(population)
[0 0 1]
[0 1 1]
[0 1 1]
...
[0 1 1]
[1 1 1]
```

Note that this process is stochastic:

```
>>> np.random.seed(2)
>>> generations = game.moran_process(initial_population=(0, 0, 1))
>>> for population in generations:
...     print(population)
[0 0 1]
[0 0 1]
[0 0 0]
```

It is possible to pass a `mutation_probability` to the process in which case it will not terminate:

```
>>> np.random.seed(2)
>>> number_of_generations = 5
>>> mutation_probability = 1
>>> generations = game.moran_process(initial_population=(0, 0, 1), mutation_
↪probability=mutation_probability)
>>> for _ in range(number_of_generations):
```

(continues on next page)

(continued from previous page)

```
...     print(next(generations))
[0 0 1]
[1 0 1]
[1 1 1]
[0 1 1]
[0 1 0]
```

Currently, only non-negative valued matrices are supported:

```
>>> A = np.array([[3, -1], [1, 2]])
>>> game = nash.Game(A)
>>> generations = game.moran_process(initial_population=(0, 0, 1))
>>> for population in generations:
...     print(population)
Traceback (most recent call last):
 ...
ValueError: Only non negative valued payoff matrices are currently supported
```

2.14 Obtain fixation probabilities

Using the implemented *Moran process <how-to-use-moran_process>* the fixation probabilities can be approximated using a method on the `Game` class:

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

The `fixation` method returns an array with the fixation probabilities of each strategy given the initial population:

```
>>> np.random.seed(0)
>>> probabilities = game.fixation_probabilities(initial_population=(0, 1, 1, 1),␣
↪repetitions=200)
>>> probabilities
array([0.235, 0.765])
```

This above shows that approximately (estimated over 200 iterations) 23.5 % of the time the first strategy will take over a population with a total of 4 individuals (when the initial population begins with 3 individuals of the other type).

2.15 Obtain a repeated game

Given a game in `Nashpy` it is possible to create a new game by repeating it as described in *Repeated Games*:

```
>>> import nashpy as nash
>>> import nashpy.repeated_games
>>> import numpy as np
>>> A = np.array([[1, -1], [-1, 1]])
>>> matching_pennies = nash.Game(A)
>>> repeated_matching_pennies = nash.repeated_games.obtain_repeated_game(game=matching_
↪pennies, repetitions=2)
```

(continues on next page)

(continued from previous page)

```
>>> repeated_matching_pennies
Zero sum game with payoff matrices:

Row player:
[[ 2.  2.  2. ...  0. -2. -2.]
 [ 2.  2.  2. ...  0. -2. -2.]
 [ 2.  2.  2. ...  0. -2. -2.]
 ...
 [ 0.  0.  0. ...  2.  0.  2.]
 [-2. -2. -2. ...  0.  2.  0.]
 [-2. -2. -2. ...  2.  0.  2.]]

Column player:
[[-2. -2. -2. ...  0.  2.  2.]
 [-2. -2. -2. ...  0.  2.  2.]
 [-2. -2. -2. ...  0.  2.  2.]
 ...
 [ 0.  0.  0. ... -2.  0. -2.]
 [ 2.  2.  2. ...  0. -2.  0.]
 [ 2.  2.  2. ... -2.  0. -2.]]
```

Note, that these games can become large even for small values of `repetitions`. The above game has payoff matrices with size:

```
>>> repeated_matching_pennies.payoff_matrices[0].shape
(32, 32)
```

When studying these large games direct computation of equilibria is unlikely to be computationally efficient. Instead learning algorithms should be used.

It is also to directly obtain a generator containing the strategies, *which are mapping from histories of play to actions*:

```
>>> strategies = nash.repeated_games.obtain_strategy_space(A=A, repetitions=2)
>>> next(strategies)
{((), ()): (1.0, 0.0), ((0,), (0,)): (1.0, 0.0), ((0,), (1,)): (1.0, 0.0), ((1,), (0,)):␣
↪(1.0, 0.0), ((1,), (1,)): (1.0, 0.0)}
```

CHAPTER

THREE

DISCUSSION

3.1 Normal Form Games

3.1.1 Motivating example: Coordination Game

Game theory is the study of interactive decision making. One example of this is the following situation:

> Two friends must decide what movie to watch at the cinema. Alice would like to watch a sport movie and Bob would like to watch a comedy. Importantly, they would both rather spend their evening together than apart.

To quantify this mathematically, numeric values are associated to the 4 possible outcomes:

1. Alice watches a sport movie, Bob watches a comedy: Alice receives a utility of 1 and Bob a utility of 1.
2. Alice watches a comedy, Bob watches a sport movie: Alice receives a utility of 0 and Bob a utility of 0.
3. Alice and Bob both watch a sport movie: Alice receives a utility of 3 and Bob a utility of 2.
4. Alice and Bob both watch a comedy: Alice receives a utility of 2 and Bob a utility of 3.

This particular example will be represented using two matrices.

A will represent the utilities of Alice:

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

B will represent the utilities of Bob

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

Alice is referred to as the row player and Bob as the column player:

- The row player chooses which row of the matrices the player will gain their utilities.
- The column player chooses which column of the matrices the player will gain their utilities.

This representation of the strategic interaction between Alice and Bob is called a *Normal Form Game*

3.1.2 Definition of Normal Form Game

An N player normal form game consists of:

- A finite set of N players.
- Action set for the players: $\{\mathcal{A}_1, \mathcal{A}_2, \dots \mathcal{A}_N\}$
- Payoff functions for the players: $u_i : \mathcal{A}_1 \times \mathcal{A}_2 \cdots \times \mathcal{A}_N \to \mathbb{R}$

Question

For the *Coordination game*:

1. What is the finite set of players?
2. What are the action sets?
3. What are the payoff functions?

Answer

1. The two players are Alice and Bob ($N = 2$).
2. The action sets are: $\mathcal{A}_1 = \mathcal{A}_2 = \{\text{Sport}, \text{Comedy}\}$
3. The payoff functions are given by the matrices A, B where the first row or column corresponds to Sport and the second row or column corresponds to Comedy.

$$u_1(\dashv_1, \dashv_2) = A_{\dashv_1, \dashv_2} \qquad u_2(\dashv_1, \dashv_2) = B_{\dashv_1, \dashv_2}$$

where $\dashv_1 \in \mathcal{A}_1$ and $\dashv_2 \in \mathcal{A}_2$.

3.1.3 Definition of a Zero Sum Game

A two player normal form game with payoff matrices A, B is called zero sum if and only if:

$$A = -B$$

Question

Is the *Coordination game* zero sum?

Answer

$A \neq -B$ so the Coordination game is not Zero sum.

3.1.4 Examples of other Normal Form Games

Prisoners Dilemma

Assume two thieves have been caught by the police and separated for questioning. If both thieves cooperate and do not divulge any information they will each get a short sentence (with a utility value of 3). If one defects they are offered a deal (utility value of 5) while the other thief will get a long sentence (utility value of 0). If they both defect they both get a medium length sentence (utility value of 1).

Question

For the Prisoners Dilemma

1. What is the finite set of players?
2. What are the action sets?
3. What are the payoff functions?
4. Is the game zero sum?

Answer

1. The two players are the two thiefs ($N = 2$).
2. The action sets are: $\mathcal{A}_1 = \mathcal{A}_2 = \{\text{Cooperate}, \text{Defect}\}$
3. The payoff functions are given by the matrices A, B where the first row or column corresponds to Cooperate and the second row or column corresponds to Defect.

$$A = \begin{pmatrix} 3 & 0 \\ 5 & 1 \end{pmatrix} \qquad B = \begin{pmatrix} 3 & 5 \\ 0 & 1 \end{pmatrix}$$

$$u_1(\dashv_1, \dashv_2) = A_{\dashv_1, \dashv_2} \qquad u_2(\dashv_1, \dashv_2) = B_{\dashv_1, \dashv_2}$$

   where $\dashv_1 \in \mathcal{A}_1$ and $\dashv_2 \in \mathcal{A}_2$.
4. The game is not Zero sum as $A \neq -B$.

Hawk Dove Game

Suppose two birds of prey must share a limited resource. The birds can act like a hawk or a dove. Hawks always act aggressively over the resource to the point of exterminating another hawk (both hawks get a utility value of 0) and/or take a majority of the resource from a dove (the hawk gets a utility value of 3 and the dove a utility value of 1). Two doves can share the resource (both getting a utility value of 2).

Question

For the Hawk Dove Game

1. What is the finite set of players?
2. What are the action sets?
3. What are the payoff functions?
4. Is the game zero sum?

Answer

1. The two players are two birds $N = 2$.
2. The action sets are: $\mathcal{A}_1 = \mathcal{A}_2 = \{\text{Hawk}, \text{Dove}\}$
3. The payoff functions are given by the matrices A, B where the first row or column corresponds to Hawk and the second row or column corresponds to Dove.

$$A = \begin{pmatrix} 0 & 3 \\ 1 & 2 \end{pmatrix} \qquad B = \begin{pmatrix} 0 & 1 \\ 3 & 2 \end{pmatrix}$$

$$u_1(\dashv_1, \dashv_2) = A_{\dashv_1, \dashv_2} \qquad u_2(\dashv_1, \dashv_2) = B_{\dashv_1, \dashv_2}$$

   where $\dashv_1 \in \mathcal{A}_1$ and $\dashv_2 \in \mathcal{A}_2$.
4. The game is not Zero sum as $A \neq -B$.

Pigs

Consider two pigs. One dominant pig and one subservient pig. These pigs share a pen. There is a lever in the pen that delivers food but if either pig pushes the lever it will take them a little while to get to the food.

- If the dominant pig pushes the lever, the subservient pig has some time to eat most of the food before being pushed out of the way. The dominant pig gets a utility value of 2 and the subservient pig gets a utility value of 3.
- If the subservient pig pushes the lever, the dominant pig will eat all the food. The dominant pig gets a utility value of 6 and the subservient pig gets a utility value of -1.
- If both pigs push the lever, the subservient pig will a small amount of the food. The dominant pig gets a utility value of 4 and the subservient pig gets a utility value of 2.
- If both pigs do not push the lever they both get a utility value of 0.

Question

For the Pigs Game

1. What is the finite set of players?
2. What are the action sets?
3. What are the payoff functions?
4. Is the game zero sum?

Answer

1. The two players are dominant and a subservient pig $N = 2$.
2. The action sets are: $\mathcal{A}_1 = \mathcal{A}_2 = \{\text{Push}, \text{Do not push}\}$
3. The payoff functions are given by the matrices A, B where the first row or column corresponds to Push and the second row or column corresponds to Do not push.

$$A = \begin{pmatrix} 4 & 2 \\ 6 & 0 \end{pmatrix} \qquad B = \begin{pmatrix} 2 & 3 \\ -1 & 0 \end{pmatrix}$$

$$u_1(\dashv_1, \dashv_2) = A_{\dashv_1, \dashv_2} \qquad u_2(\dashv_1, \dashv_2) = B_{\dashv_1, \dashv_2}$$

where $\dashv_1 \in \mathcal{A}_1$ and $\dashv_2 \in \mathcal{A}_2$.

4. The game is not Zero sum as $A \neq -B$.

Matching Pennies

Consider two players who can choose to display a coin either Heads facing up or Tails facing up. If both players show the same face then player 1 wins, if not then player 2 wins. Winning corresponds to a numeric value of 1 and losing a numeric value of -1.

Question

For the Matching Pennies game:

1. What is the finite set of players?
2. What are the action sets?
3. What are the payoff functions?
4. Is the game zero sum?

Answer

1. There are two players $N = 2$.
2. The action sets are: $\mathcal{A}_1 = \mathcal{A}_2 = \{\text{Heads}, \text{Tails}\}$
3. The payoff functions are given by the matrices A, B where the first row or column corresponds to Heads and the second row or column corresponds to Tails.

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

$$u_1(\dashv_1, \dashv_2) = A_{\dashv_1, \dashv_2} \qquad u_2(\dashv_1, \dashv_2) = B_{\dashv_1, \dashv_2}$$

where $\dashv_1 \in \mathcal{A}_1$ and $\dashv_2 \in \mathcal{A}_2$.

4. The game is Zero sum as $A = -B$.

3.1.5 Using Nashpy

See *Create a Normal Form Game* for guidance of how to use Nashpy to create a Normal form game.

3.2 Strategies

3.2.1 Motivating example: Strategy for Rock Paper Scissors

The game of Rock Paper Scissors is a common parlour game between two players who pick 1 of 3 options simultaneously:

1. Rock which beats Scissors;
2. Paper which beats Rock;
3. Scissors which beats Paper

Thus, this corresponds to a Normal Form Game with:

1. Two players ($N = 2$).
2. The action sets are: $\mathcal{A}_1 = \mathcal{A}_2 = \{\text{Rock}, \text{Paper}, \text{Scissors}\}$
3. The payoff functions are given by the matrices A, B where the first row or column corresponds to Rock, the second to Paper and the third to Scissors.

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

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

If we consider two players, assume the row player always chooses Paper and the column player randomly chooses from Rock and Paper (with equal probability) what is the expected outcome of any one game between them?

- The expected score of the row player will be: $-1 \times 1/2 + 0 \times 1/2 = -1/2$.
- The expected score of the column player will be: $1 \times 1/2 + 0 \times 1/2 = 1/2$.

In Game theoretic terms, the behaviours described above are referred to as **strategies**. **Strategies map information to actions.** In this particular case, the only available information is the game itself and the actions are $\mathcal{A}_1 = \mathcal{A}_2$.

3.2.2 Definition of a strategy in a normal form game

A strategy for a player with action set $\mathcal{A}$ is a probability distribution over elements of $\mathcal{A}$.

Typically a strategy is denoted by $\sigma \in [0, 1]^{|\mathcal{A}|}_{\mathbb{R}}$ so that:

$$\sum_{i=1}^{\mathcal{A}} \sigma_i = 1$$

Question

For *Rock Papoer Scissors*:

1. What is the strategy σ_r that corresponds to the row player's behaviour of always choosing Paper?
2. What is the strategy σ_c that corresponds to the column player's behaviour of always randomly choosing between Rock and Paper?

Answer

1. $\sigma_r = (0, 1, 0)$
2. $\sigma_c = (1/2, 1/2, 0)$

3.2.3 Definition of support of a strategy

For a given strategy σ, the support of σ: $\mathcal{S}(\sigma)$ is the set of actions $i \in \mathcal{A}$ for which $\sigma_i > 0$.

Question

For the following strategies σ obtain $\mathcal{S}(\sigma)$:

1. $\sigma = (1, 0, 0)$
2. $\sigma = (1/3, 1/3, 1/3)$
3. $\sigma = (2/5, 0, 3/5)$

Answer

1. $\mathcal{S}(\sigma) = \{1\}$
2. $\mathcal{S}(\sigma) = \{1, 2, 3\}$
3. $\mathcal{S}(\sigma) = \{1, 3\}$

Note here that as no specific action sets are given the integers are used.

3.2.4 Strategy spaces for Normal form Games

Given a set of actions $\mathcal{A}$ the space of all strategies $\mathcal{S}$ is defined as:

$$\mathcal{S} = \left\{ \sigma \in [0, 1]^{|\mathcal{A}|}_{\mathbb{R}} \;\middle|\; \sum_{i=1}^{\mathcal{A}} \sigma_i = 1 \right\}$$

3.2.5 Calculation of expected utilities

Considering a game $(A, B) \in \mathbb{R}^{(m \times n)^2}$, if σ_r and σ_c are the strategies for the row/column player, the expected utilities are:

- For the row player: $u_r(\sigma_r, \sigma_c) = \sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij} \sigma_{r_i} \sigma_{c_j}$
- For the column player: $u_c(\sigma_r, \sigma_c) = \sum_{i=1}^{m} \sum_{j=1}^{n} B_{ij} \sigma_{r_i} \sigma_{c_j}$

This corresponds to taking the expectation over the probability distributions σ_r and σ_c.

Question

For the *Rock Papoer Scissors*:

What are the expected utilities to both players if $\sigma_r = (1/3, 0, 2/3)$ and $\sigma_c = (1/3, 1/3, 1/3)$.

Answer

$$
\begin{aligned}
u_r(\sigma_r, \sigma_c) =& 1/3(1/3 \times 0 + 1/3 \times -1 + 1/3 \times 1) && (3.1)\\
&+ 0(1/3 \times 1 + 1/3 \times 0 + 1/3 \times && (3.2)\\
&+ 2/3(1/3 \times -1 + 1/3 \times 1 + 1/3 && (3.3)\\
& && (3.4)
\end{aligned}
$$

$$
\begin{aligned}
u_c(\sigma_r, \sigma_c) =& 1/3(1/3 \times 0 + 1/3 \times 1 + 1/3 \times -1) && (3.5)\\
&+ 0(1/3 \times -1 + 1/3 \times 0 + 1/3 && (3.6)\\
&+ 2/3(1/3 \times 1 + 1/3 \times -1 + 1/3 && (3.7)\\
& && (3.8)
\end{aligned}
$$

3.2.6 Linear algebraic calculation of expected utilities

Given a game $(A, B) \in \mathbb{R}^{(m \times n)^2}$, considering σ_r and σ_c as vectors in $\mathbb{R}^m$ and $\mathbb{R}^n$. The expected utilities can be written as the matrix vector product:

- For the row player: $u_r(\sigma_r, \sigma_c) = \sigma_r A \sigma_c^T$
- For the column player: $u_c(\sigma_r, \sigma_c) = \sigma_r B \sigma_c^T$

Question

For *Rock Paper Scissors*:

Calculate the expected utilities to both players if $\sigma_r = (1/3, 0, 2/3)$ and $\sigma_c = (1/3, 1/3, 1/3)$ using a linear algebraic approach.

Answer

$$u_r(\sigma_r, \sigma_c) = (1/3, 0, 2/3) A \begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix} = (-2/3, 1/3, 1/3) \begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix} = 0$$

$$u_c(\sigma_r, \sigma_c) = (1/3, 0, 2/3) B \begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix} = (2/3, -1/3, -1/3) \begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix} = 0$$

3.2.7 Using Nashpy

See *Calculate utilities* for guidance of how to use Nashpy to calculate utilities.

3.3 Best responses

3.3.1 Motivating example: Best Responses in Matching Pennies

Considering the game *Matching Pennies*:

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

If the row player knows that the column player is playing the *strategy* $\sigma_c = (0, 1)$ the utility of the row player is maximised by playing $\sigma_r = (0, 1)$.

In this case σ_r is referred to as a **best response** to σ_c.

Alternatively, if the column player knows that the row player is playing the *strategy* $\sigma_r = (0, 1)$ the column player's best response is $\sigma_c = (1, 0)$.

3.3.2 Definition of a best response in a normal form game

In a two player game $(A, B) \in \mathbb{R}^{m \times n^2}$ a strategy σ_r^* of the row player is a best response to a column players' strategy σ_c if and only if:

$$\sigma_r^* = \text{argmax}_{\sigma_r \in \mathcal{S}_1} \sigma_r A \sigma_c^T.$$

Where $\mathcal{S}_1$ denotes the *space of all strategies* for the first player.

Similarly a mixed strategy σ_c^* of the column player is a best response to a row players' strategy σ_r if and only if:

$$\sigma_c^* = \text{argmax}_{\sigma_c \in \mathcal{S}_2} \sigma_r B \sigma_c^T.$$

Question

For the *Prisoners Dilemma*:

What is the row player's best response to either of the actions of the column player?

Answer

Recalling that A is given by:

$$A = \begin{pmatrix} 3 & 0 \\ 5 & 1 \end{pmatrix}$$

Against the first action of the column player the best response is to choose the second action which gives a utility of 5. This can be expressed as:

$$\text{argmax}_{i \in \mathcal{S}_1} A_{i1} = 2$$

Against the second action of the column player the best response is to choose the second action which gives a utility of 1. This can be expressed as:

$$\text{argmax}_{i \in \mathcal{S}_1} A_{i2} = 2$$

The row player's best response to either of the actions of the column player is $\sigma_r^* = (1, 0)$. This can be expressed as:

$$\text{argmax}_{i \in \mathcal{S}_1} A_{ij} = 2 \text{ for all } j \in \mathcal{A}_2$$

3.3.3 Generic best responses in 2 by 2 games

In two player normal form games with $|A_1| = |A_2| = 2$: a 2 by 2 game, the utility of a row player playing $\sigma_r = (x, 1-x)$ against a strategy $\sigma_c = (y, 1-y)$ is linear in x:

$$\begin{aligned} u_r(\sigma_r, \sigma_c) &= (x, 1-x)A(y, 1-y)^T \\ &= A_{11}xy + A_{12}x(1-y) + A_{21}(1-x)y + A_{22}(1-x)(1-y) \\ &= ax + b \end{aligned}$$

where:

$$\begin{aligned} a &= A_{11}y + A_{12}(1-y) - A_{21}y - A_{22}(1-y) \\ b &= A_{21}y + A_{22}(1-y) \end{aligned}$$

This observation allows us to obtain the best response σ_r^* against any $\sigma_c = (y, 1-y)$.

For example, consider *Matching Pennies*. Below is a plot of $u_r(\sigma_r, \sigma_c)$ as a function of y for $\sigma_r \in \{(1, 0), (0, 1)\}$.

Given that the utilities in both cases are linear, the best response to any value of $y \neq 1/2$ is either $(1, 0)$ or $(0, 1$. The best response σ_r^* is given by:

$$\sigma_r^* = \begin{cases} (1, 0), & \text{if } y > 1/2 \\ (0, 1), & \text{if } y < 1/2 \\ \text{indifferent}, & \text{if } y = 1/2 \end{cases}$$

Question

For the *Matching Pennies* game:

What is the column player's best response as a function of x where $\sigma_r = (x, 1-x)$.

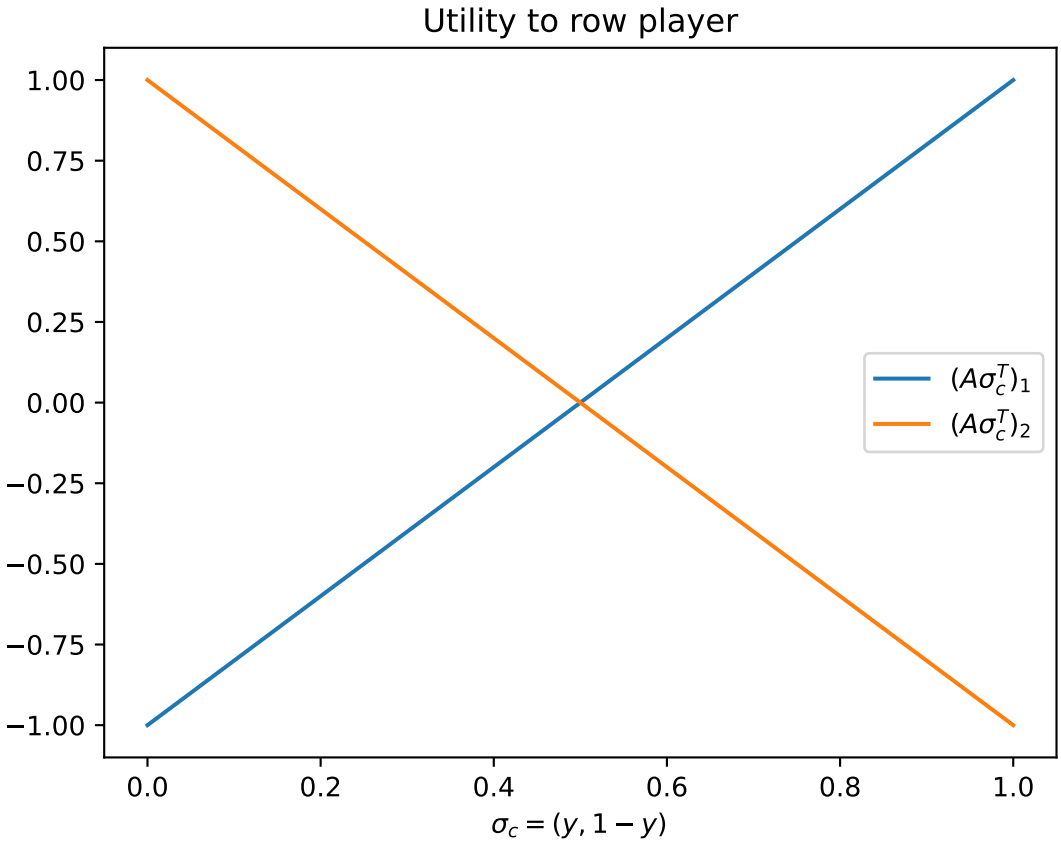
Answer

Recalling that B is given by:

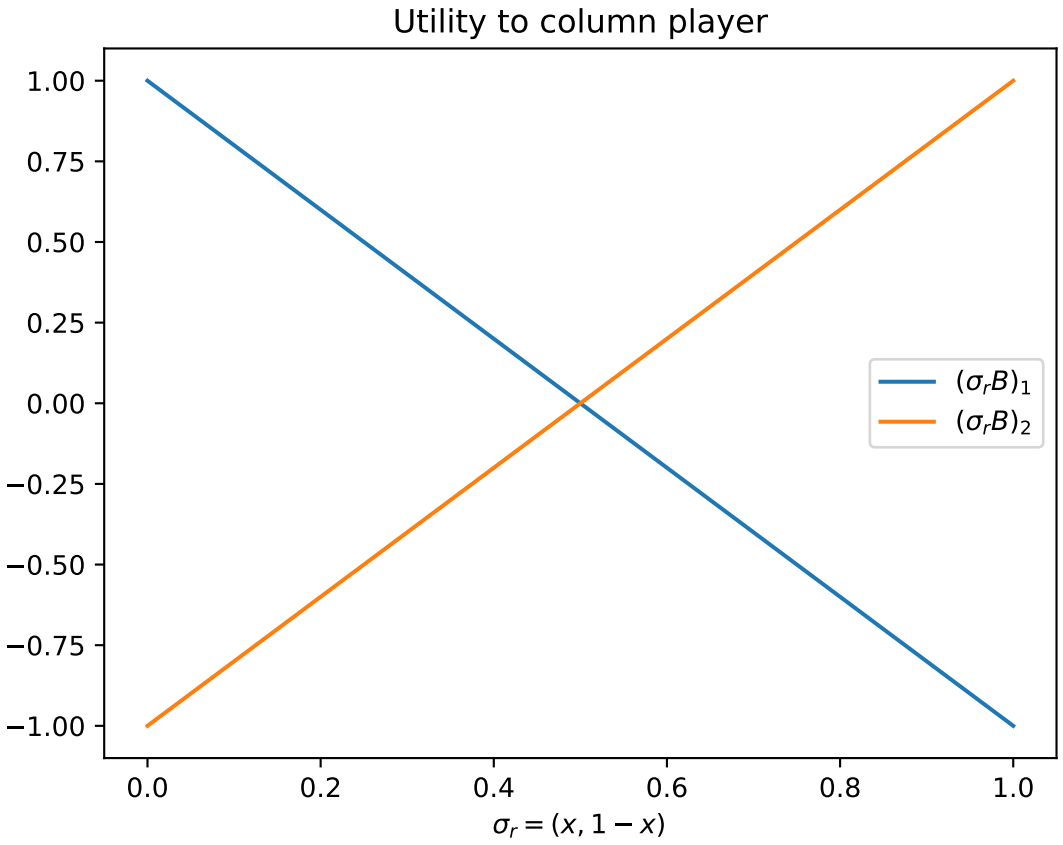
$$B = \begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix}$$

This gives:

$$\begin{aligned} u_c(\sigma_r, (1, 0)) &= -x + (1-x) = 1 - 2x \\ &= x - (1-x) = -1 + 2x \end{aligned}$$


Utility to row player
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.2
0.4
0.6
0.8
1.0
$\sigma_c = (y, 1-y)$
$(A\sigma_c^T)_1$
$(A\sigma_c^T)_2$

Here is a plot of the utilities:



3.3.4 General condition for a best response

In a two player game $(A, B) \in \mathbb{R}^{m \times n^2}$ a strategy σ_r^* of the row player is a best response to a column players' strategy σ_c if and only if:

$$\sigma_{r^* i} > 0 \Rightarrow (A\sigma_c^T)_i = \max_{k \in \mathcal{A}_2} (A\sigma_c^T)_k \text{ for all } i \in \mathcal{A}_1$$

Proof

$(A\sigma_c^T)_i$ is the utility of the row player when they play their i^{th} action. Thus:

$$\sigma_r A \sigma_c^T = \sum_{i=1}^{m} \sigma_{ri} (A\sigma_c^T)_i$$

Let $u = \max_k(A\sigma_c^T)_k$ giving:

$$\begin{aligned}\sigma_r A\sigma_c^T &= \sum_{i=1}^{m} \sigma_{ri}(u - u + (A\sigma_c^T)_i) \\ &= \sum_{i=1}^{m} \sigma_{ri}u - \sum_{i=1}^{m} \sigma_{ri}(u - (A\sigma_c^T)_i) \\ &= u - \sum_{i=1}^{m} \sigma_{ri}(u - (A\sigma_c^T)_i)\end{aligned}$$

We know that $u - (A\sigma_c^T)_i \geq 0$, thus the largest $\sigma_r A\sigma_c^T$ can be is u which occurs if and only if $\sigma_{ri} > 0 \Rightarrow (A\sigma_c^T)_i = u$ as required.

Question

For the *Rock Paper Scissors* game:

Which of the following pairs of strategies are best responses to each other:

1. $\sigma_r = (0, 0, 1)$ and $\sigma_c = (0, 1/2, 1/2)$
2. $\sigma_r = (1/3, 1/3, 1/3)$ and $\sigma_c = (0, 1/2, 1/2)$
3. $\sigma_r = (1/3, 1/3, 1/3)$ and $\sigma_c = (1/3, 1/3, 1/3)$

Answer

Recalling that A and B are given by:

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

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

We can apply the best response condition to each pairs of strategies:

1. $A\sigma_c^T = \begin{pmatrix} 0 \\ -1/2 \\ 1/2 \end{pmatrix}$. $\max(A\sigma_c^T) = 1/2$. The only i for which $\sigma_{ri} > 0$ is $i = 3$ and $(A\sigma_c^T)_3 = \max(A\sigma_c^T)$ thus σ_r **is a best response to** σ_c. $\sigma_r B = (1, -1, 0)$. $\max(\sigma_r B) = 1$. The values of i for which $\sigma_{ci} > 0$ are $i = 2$ and $i = 3$ but $(\sigma_r B)_2 \neq \max(\sigma_r B)$ thus σ_c **is not a best response to** σ_r.
2. $A\sigma_c^T = \begin{pmatrix} 0 \\ -1/2 \\ 1/2 \end{pmatrix}$. $\max(A\sigma_c^T) = 1/2$. The values of i for which $\sigma_{ri} > 0$ are $i = 1$, $i = 2$ and $i = 3$ however, $(A\sigma_c^T)_2 \neq \max(A\sigma_c^T)$ thus σ_r **is not a best response to** σ_c. $\sigma_r B = (0, 0, 0)$. $\max(\sigma_r B) = 0$. The values of i for which $\sigma_{ci} > 0$ are $i = 2$ and $i = 3$ and $(\sigma_r B)_2 = (\sigma_r B)_3 = \max(\sigma_r B)$ thus σ_c **is a best response to** σ_r.
3. $A\sigma_c^T = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$. $\max(A\sigma_c^T) = 0$. The values of i for which $\sigma_{ri} > 0$ are $i = 1$, $i = 2$ and $i = 3$ and $(A\sigma_c^T)_1 = (A\sigma_c^T)_2 = (A\sigma_c^T)_3 = \max(A\sigma_c^T)$ thus σ_r **is a best response to** σ_c. $\sigma_r B = (0, 0, 0)$. $\max(\sigma_r B) = 0$. The values of i for which $\sigma_{ci} > 0$ are $i = 1$, $i = 2$ and $i = 3$ and $(\sigma_r B)_1 = (\sigma_r B)_2 = (\sigma_r B)_3 = \max(\sigma_r B)$ thus σ_c **is a best response to** σ_r.

3.3.5 Definition of Nash equilibrium

In a two player game $(A, B) \in \mathbb{R}^{m \times n^2}$, (σ_r, σ_c) is a Nash equilibria if σ_r is a best response to σ_c and σ_c is a best response to σ_r.

3.3.6 Using Nashpy

See *Check if a strategy is a best response* for guidance of how to use Nashpy to check if a strategy is a best response.

3.4 Support enumeration

3.4.1 Motivating example: Coordination Game

In the *Coordination game* in how many situations do neither player have an incentive to **independently** change their strategy?

Neither player having a reason to change their strategy implies that both strategies are *Best responses* to each other.

To identify such pairs of strategies, we will use the *General condition for a best response* by considering all possible non zero valued elements σ_r and σ_c.

Recall that for the Coordination game the matrices A and B are given by:

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

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

If we consider strategies that only play a single action there are two options for each strategy:

$$\sigma_r \in \{(1, 0), (0, 1)\}$$

and:

$$\sigma_c \in \{(1, 0), (0, 1)\}$$

We will inspect all four combinations:

- $\sigma_r = (1, 0)$ and $\sigma_c = (1, 0)$ which corresponds to both players playing their first action which gives: $u_r(\sigma_r, \sigma_c) = 3$ and $u_c(\sigma_r, \sigma_c) = 2$. If the row player where to modify their strategy (while the column player stayed unchanged) to play the second action their utility would decrease. Likewise, if the column player were to modify their strategy their utility would also decrease.
- $\sigma_r = (1, 0)$ and $\sigma_c = (0, 1)$ which corresponds to the row player playing their first action and the column player playing their second action which gives: $u_r(\sigma_r, \sigma_c) = 1$ and $u_c(\sigma_r, \sigma_c) = 1$. In this case, if either player were to move their utility would increase.
- $\sigma_r = (0, 1)$ and $\sigma_c = (1, 0)$ which corresponds to the row player playing their second action and the column player playing their first action which gives: $u_r(\sigma_r, \sigma_c) = 0$ and $u_c(\sigma_r, \sigma_c) = 0$. In this case, if either player were to move their utility would increase.
- $\sigma_r = (0, 1)$ and $\sigma_c = (0, 1)$ which corresponds to both players playing their second action which gives: $u_r(\sigma_r, \sigma_c) = 2$ and $u_c(\sigma_r, \sigma_c) = 3$. If the row player where to modify their strategy (while the column player stayed unchanged) to play the second action their utility would decrease. Likewise, if the column player were to modify their strategy their utility would also decrease.

If we now consider strategies that play **both** actions there is a single general form:

$$\sigma_r = (x, 1-x) \text{ for } 0 < x < 1$$

$$\sigma_c = (y, 1-y) \text{ for } 0 < y < 1$$

We can apply the *General condition for a best response* here.

If σ_r is a best response to σ_c then:

$$(A\sigma_c T)_i = \max_{k \in \{1,2\}} (A\sigma_c^T)_k \text{ for all } i \in \{1, 2\}$$

which gives:

$$\begin{aligned} 3y + 1(1-y) &= \max_{k \in \{1,2\}} (A\sigma_c^T)_k \\ 0y + 2(1-y) &= \max_{k \in \{1,2\}} (A\sigma_c^T)_k \end{aligned}$$

which in turn corresponds to:

$$\begin{aligned} 3y + 1(1-y) &= 2(1-y) \\ y &= 1/4 \end{aligned}$$

Thus $\sigma_r = (x, 1-x)$ with $0 < x < 1$ is a best response to σ_c if and only if $\sigma_c = (1/4, 3/4)$.

We will now apply the *General condition for a best response* again but to the column player:

If σ_c is a best response to σ_r then:

$$(\sigma_r B)_j = \max_{k \in \{1,2\}} (\sigma_r B)_k \text{ for all } j \in \{1, 2\}$$

which gives:

$$\begin{aligned} 2x + 0(1-x) &= \max_{k \in \{1,2\}} (\sigma_r B)_k \\ 1x + 3(1-x) &= \max_{k \in \{1,2\}} (\sigma_r B)_k \end{aligned}$$

which in turn corresponds to:

$$\begin{aligned} 2x &= x + 3(1-x) \\ x &= 3/4 \end{aligned}$$

Thus $\sigma_c = (y, 1-y)$ with $0 < y < 1$ is a best response to σ_r if and only if $\sigma_r = (3/4, 1/4)$.

There are 3 pairs of strategies that are best responses to each other:

- $\sigma_r = (1, 0)$ and $\sigma_c = (1, 0)$.
- $\sigma_r = (0, 1)$ and $\sigma_c = (0, 1)$.
- $\sigma_r = (3/4, 1/4)$ and $\sigma_c = (1/4, 3/4)$.

3.4.2 The support enumeration algorithm

The approach used in *Motivating example: Coordination Game* is in fact an application of a formalised algorithm called support enumeration.

The algorithm is as follows:

For a non *Degenerate* 2 player game $(A, B) \in \mathbb{R}^{m \times n^2}$ the following algorithm returns all pairs of best responses:

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):

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

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

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

Question

Use support enumeration to find all Nash equilibria for the game given by $A = \begin{pmatrix} 1 & 1 & -1 \\ 2 & -1 & 0 \end{pmatrix}$ and $B = \begin{pmatrix} 1/2 & -1 & -1/2 \\ -1 & 3 & 2 \end{pmatrix}$.

Answer

1. It is immediate to note that there are no pairs of pure best responses.
2. All possible support pairs are:
   - $I = \{1, 2\}$ and $J = \{1, 2\}$
   - $I = \{1, 2\}$ and $J = \{1, 3\}$
   - $I = \{1, 2\}$ and $J = \{2, 3\}$
3. Let us solve the corresponding linear equations:
   - $I = \{1, 2\}$ and $J = \{1, 2\}$:

$$1/2\sigma_{r1} - \sigma_{r2} = -\sigma_{r1} + 3\sigma_{r2}$$

$$\sigma_{r1} = 8/3\sigma_{r2}$$

$$\sigma_{c1} + \sigma_{c2} = 2\sigma_{c1} - \sigma_{c2}$$

$$\sigma_{c1} = 2\sigma_{c2}$$

   - $I = \{1, 2\}$ and $J = \{1, 3\}$:

$$1/2\sigma_{r1} - \sigma_{r2} = -1/2\sigma_{r1} + 2\sigma_{r2}$$

$$\sigma_{r1} = 3\sigma_{r2}$$

$$\sigma_{c1} - \sigma_{c3} = 2\sigma_{c1} + 0\sigma_{c3}$$

$$\sigma_{c1} = -\sigma_{c3}$$

- $I = \{1, 2\}$ and $J = \{2, 3\}$:

$$-\sigma_{r1} + 3\sigma_{r2} = -1/2\sigma_{r1} + 2\sigma_{r2}$$

$$\sigma_{r1} = 2\sigma_{r2}$$

$$\sigma_{c2} - \sigma_{c3} = -\sigma_{c2} + 0\sigma_{c3}$$

$$2\sigma_{c2} = \sigma_{c3}$$

4. We check which supports give valid strategies:

   - $I = \{1, 2\}$ and $J = \{1, 2\}$:

$$\sigma_r = (8/11, 3/11)$$

$$\sigma_c = (2/3, 1/3, 0)$$

   - $I = \{1, 2\}$ and $J = \{1, 3\}$:

$$\sigma_r = (3/4, 1/4)$$

$$\sigma_c = (k, 0, -k)$$

     which is not a valid strategy.

   - $I = \{1, 2\}$ and $J = \{2, 3\}$:

$$\sigma_r = (2/3, 1/3)$$

$$\sigma_c = (0, 1/3, 2/3)$$

5. Let us verify the best response condition:

   - $I = \{1, 2\}$ and $J = \{1, 2\}$:

$$\sigma_c = (2/3, 1/3, 0)$$

$$A\sigma_c^T = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

     Thus σ_r is a best response to σ_c

$$\sigma_r = (8/11, 3/11)$$

$$\sigma_r B = (1/11, 1/11, 2/11)$$

     Thus σ_c is not a best response to σ_r (because there is a better response outside of the support of σ_c).

   - $I = \{1, 2\}$ and $J = \{2, 3\}$:

$$\sigma_c = (0, 1/3, 2/3)$$

$$A\sigma_c^T = \begin{pmatrix} -1/3 \\ -1/3 \end{pmatrix}$$

     Thus σ_r is a best response to σ_c

$$\sigma_r = (2/3, 1/3)$$

$$\sigma_r B = (0, 1/3, 1/3)$$

     Thus σ_c is a best response to σ_r.

     Thus the (unique) Nash equilibrium for this game is:

$$((2/3, 1/3), (0, 1/3, 2/3))$$

3.4.3 Using Nashpy

See *Solve with support enumeration* for guidance of how to use Nashpy to use support enumeration.

3.5 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.

3.5.1 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:

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

   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 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 are 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)
>>> 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})
```

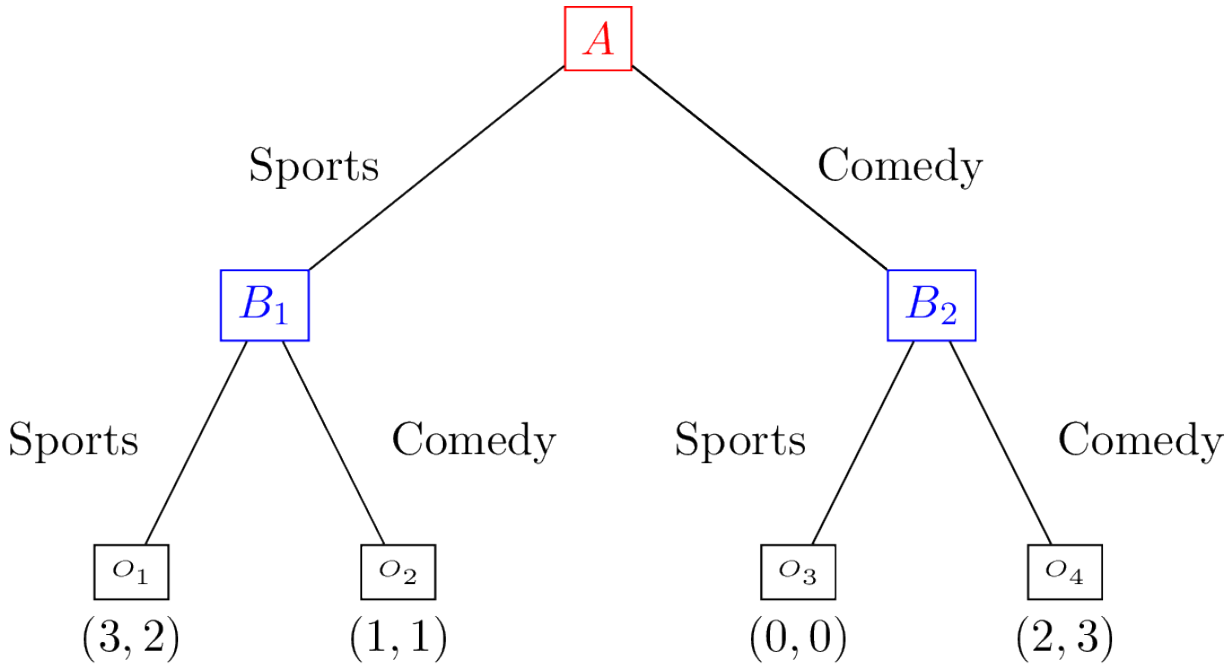
3. Step 3, we iterate over all pairs of the vertices of both polytopes and pick out the ones that are fully labeled. Because of the scaling that took place to create the Polytope from the Polyhedron, we will need to return a normalisation of both vertices.

3.6 Extensive Form Games

3.6.1 Motivating example: A modification of the Coordination Game

Consider the *Coordination game* with the modification that Alice and Bob have more information available to them: Alice decides where they are going and then lets Bob know before Bob makes their own choice.

This can be represented pictorially as follows:



3.6.2 Definition of an Extensive Form Game

An extensive form game consists of:

- A finite set of players $\mathcal{N}$.
- A tree: $G = (V, E, x^0)$ where: V is the set of vertices, E the set of edges and $x^0 \in V$ is the root of the tree.
- $(V_i)_{i \in \mathcal{N}}$ is a partition of the set of vertices that are not leaves.
- O is the set of possible game outcomes.
- u is a function mapping every leaf of G to an element of O.

Question

For the *modified coordination game*:

1. What is the finite set of players $\mathcal{N}$?
2. What the elements $G = (V, E, x^0)$?
3. What is the partition $(V_i)_{i \in \mathcal{N}}$?
4. What is the set of possible game outcomes O?
5. What is the mapping u from every leaf of G to an element of O?

Answer

1. The set $\mathcal{N}$ has two players: Alice and Bob.
2. The tree is given by:

$$V = \{A, B_1, B_2, O_1, O_2, O_3, O_4\}$$

$$E = \{(A, B_1), (A, B_2), (B_1, O_1), (B_1, O_2), (B_2, O_3), (B_2, O_4)\}$$

$$x^0 = A$$

3. The partition of of non leaf vertices is given by:

$$V_{\text{Alice}} = \{A_1\} \qquad V_{\text{Bob}} = \{B_1, B_2\}$$

4. The set of possible game ouctomes $O = \{(3, 2), (1, 1), (0, 0), (2, 3)\}$.
5. The mapping u is given by:

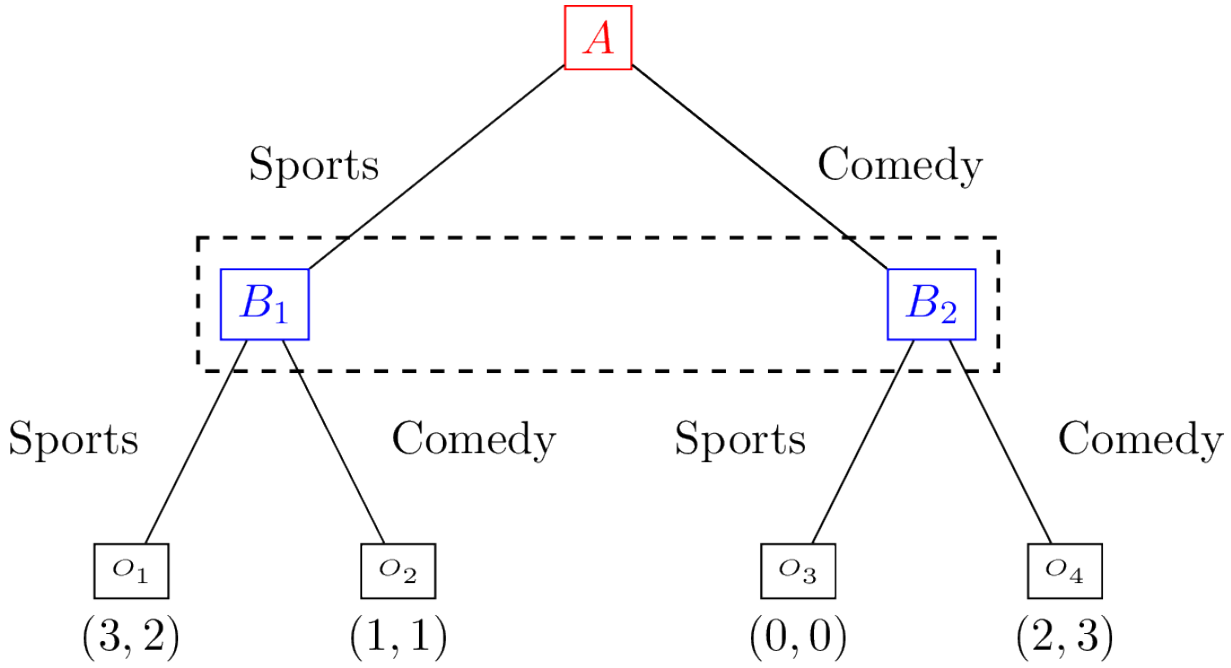
$$u(O_1) = (3, 2) \qquad u(O_2) = (1, 1) \qquad u(O_3) = (0, 0) \qquad u(O_4) = (2, 3)$$

3.6.3 Imperfect information

The modified coordination game described *here* differs from the example given in the *normal for game chapter* in that Bob knows what action is chosen by Alice.

To represent imperfect information we can partition the vertices of a game tree to indicate which vertices have the same information.

This can be represented pictorially as follows:



This indicates that Bob makes a decision at both nodes in $\{B_1, B_2\}$ without knowing at which of the two vertices they are. The set $\{B_1, B_2\}$ is called an information set.

3.6.4 Definition of an information set

Given a game in extensive form: $(\mathcal{N}, G, (V_i)_{i \in \mathcal{N}}, O, u)$ the set of information sets v_i of player $i \in \mathcal{N}$ is a partition of V_i. Each element of v_i denotes a set of nodes at which a player is unable to distinguish when choosing an action.

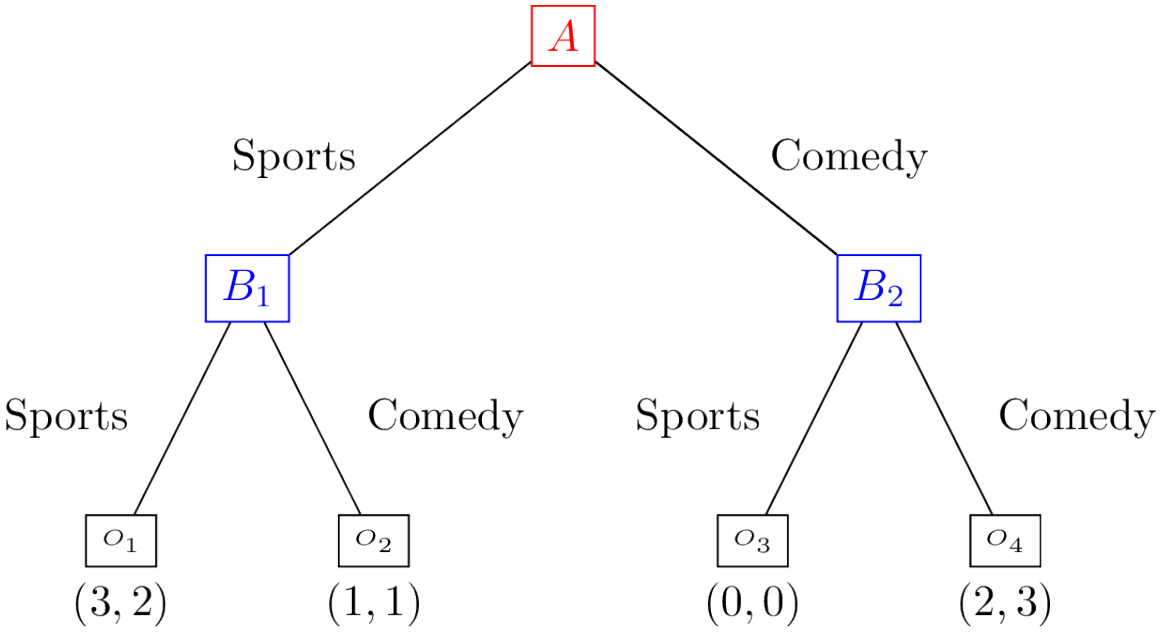
This implies that:

- Every information set contains vertices for a single player.
- All vertices in an information set must have the same number of successors (with the same action labels).

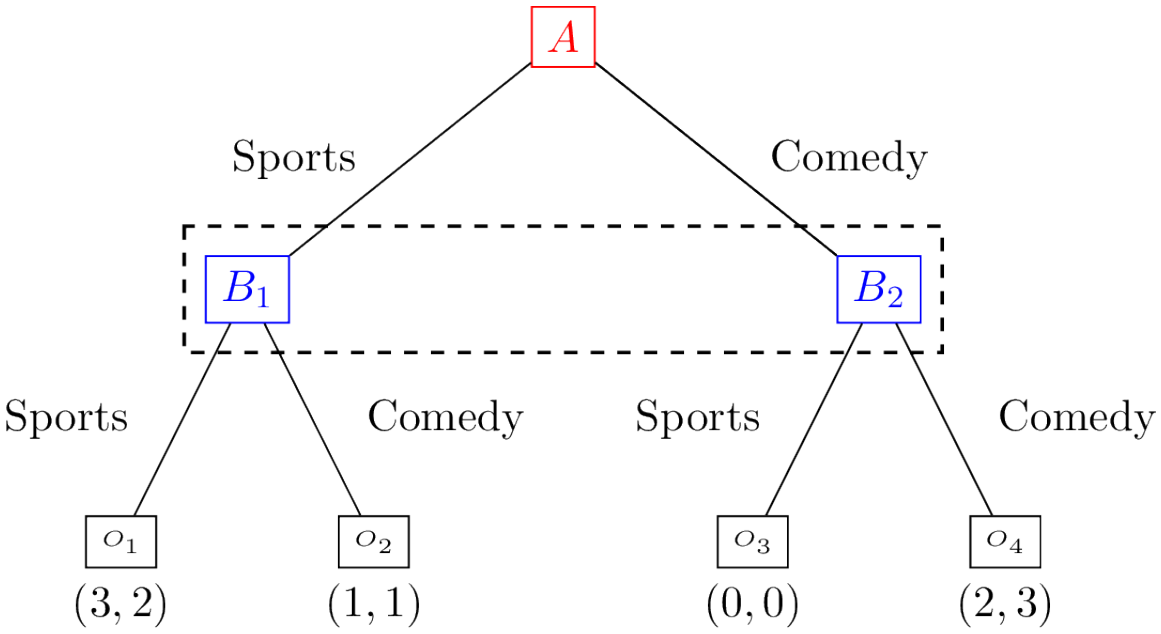
Question

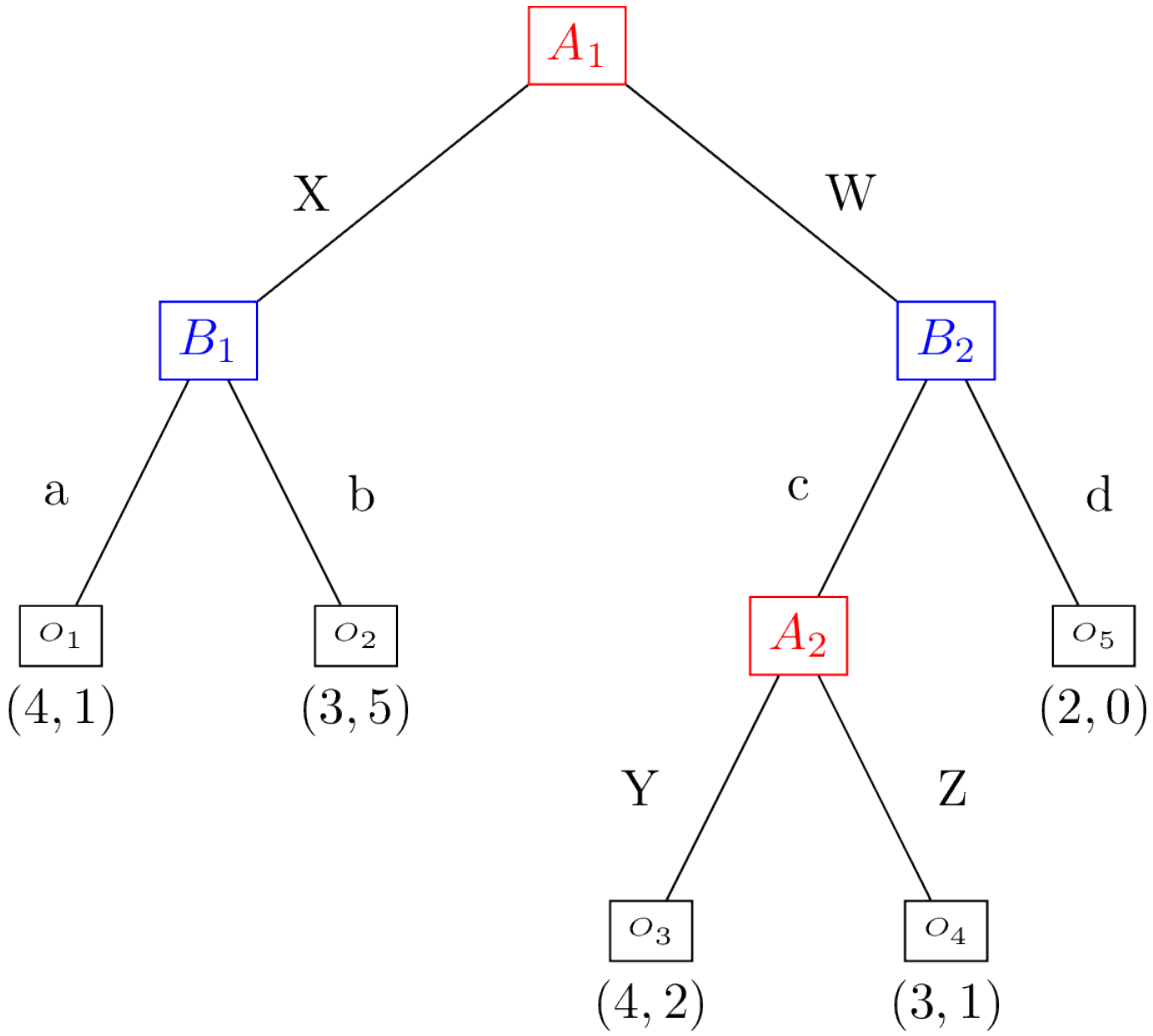
For the following games with $\mathcal{N} = \{\text{Alice}, \text{Bob}\}$, assume that decision nodes A_i are Alice's and B_i are Bob's. Obtain all information sets:

1.

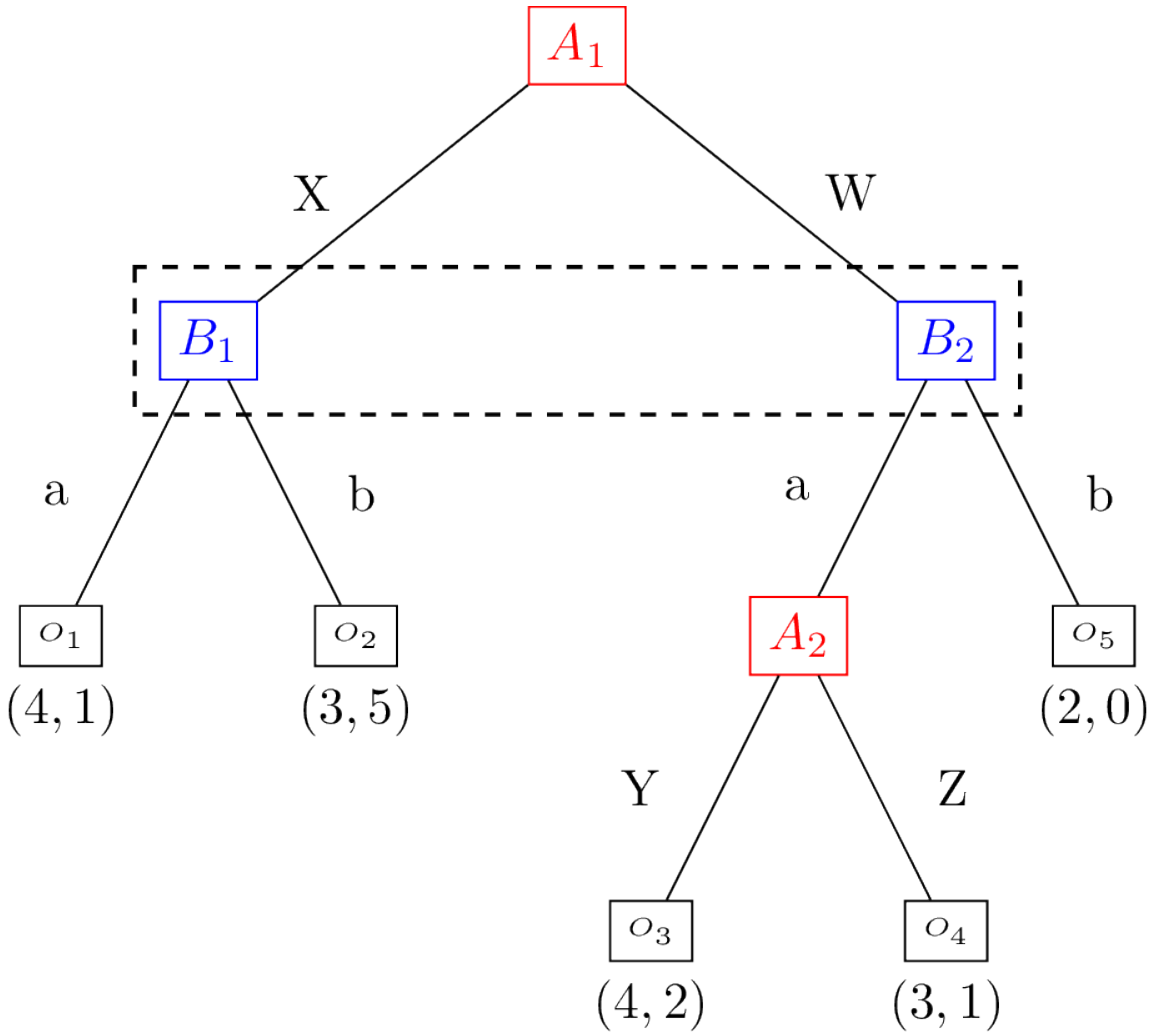

A
Sports
Comedy
B1
B2
Sports
Comedy
Sports
Comedy
o1
o2
o3
o4
(3, 2)
(1, 1)
(0, 0)
(2, 3)

2.

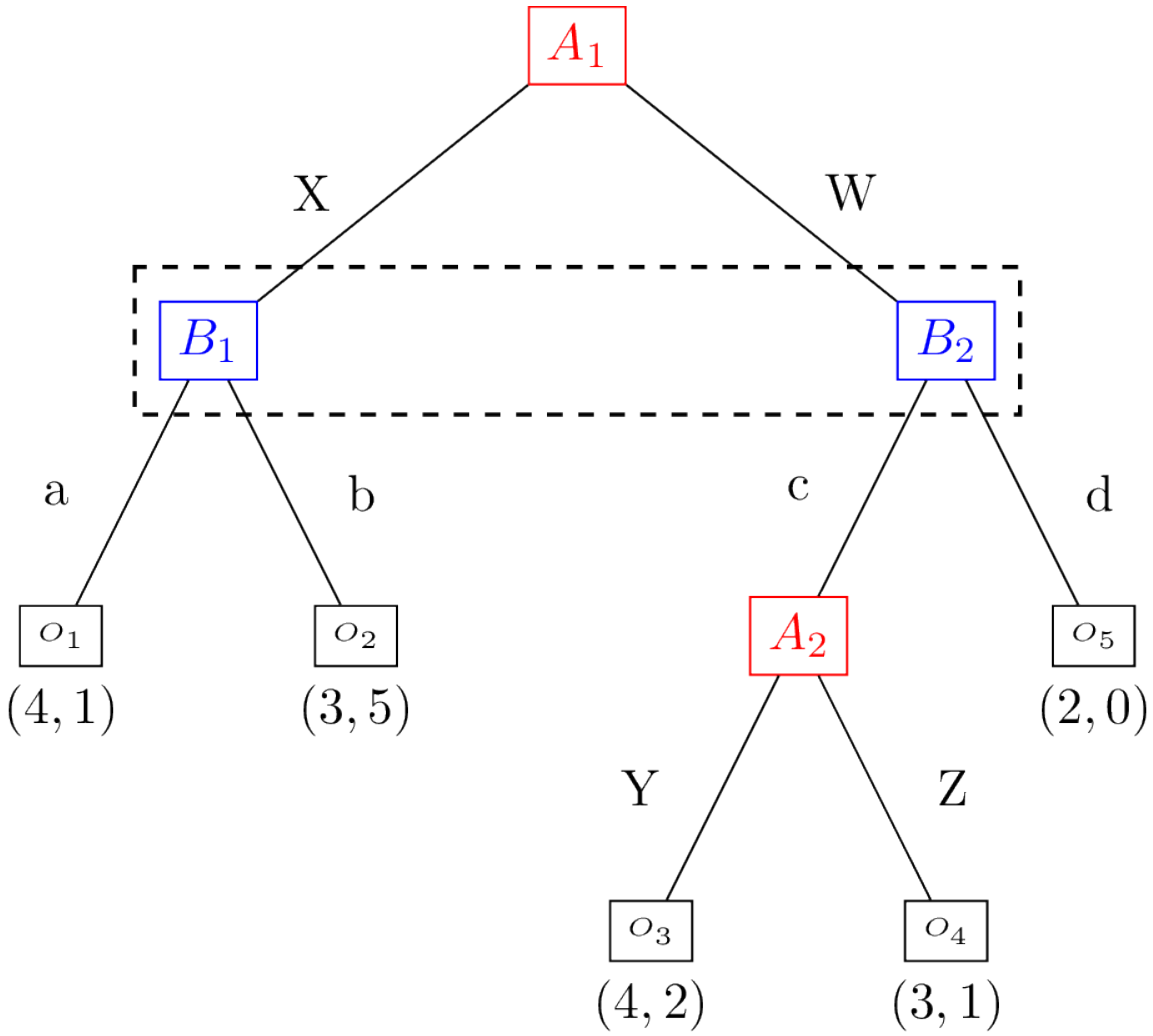

A
Sports
Comedy
B1
B2
Sports
Comedy
Sports
Comedy
o1
o2
o3
o4
(3, 2)
(1, 1)
(0, 0)
(2, 3)



3.



4.



5.

Answer

1. $v_{\text{Alice}} = \{\{A\}\}$ $v_{\text{Bob}} = \{\{B_1\}, \{B_2\}\}$
2. $v_{\text{Alice}} = \{\{A\}\}$ $v_{\text{Bob}} = \{\{B_1, B_2\}\}$
3. $v_{\text{Alice}} = \{\{A_1\}, \{A_2\}\}$ $v_{\text{Bob}} = \{\{B_1\}, \{B_2\}\}$
4. $v_{\text{Alice}} = \{\{A_1\}, \{A_2\}\}$ $v_{\text{Bob}} = \{\{B_1, B_2\}\}$
5. This game has incoherent information sets: the two vertices B_1 and B_2 have different actions.

3.6.5 Definition of a strategy in an extensive form game

A strategy for a player in an extensive form is a collection of probability distribution over the action set of each information set.

3.6.6 Equivalence of Extensive and Normal Form Games

A game in extensive form can be mapped to a game in normal form by enumerating all possible strategies that indicate single actions at each information set. This set of possible strategies corresponds to the actions in the normal form game.

These strategies can be thought of as vectors in the space of the cross product of the sets of actions available at every information set. For player $i \in \mathcal{N}$ with information sets $v_i = ((v_i)_1, (v_i)_2, \ldots, (v_i)_n)$ a strategy $s = (s_1, s_2, \ldots, s_n$ indicates what action to take at each information set. So s_2 will prescribe which action to take at all vertices contained in $(v_i)_2$.

As an example consider the *modified coordination game*. The full enumeration of strategies that indicate single actions for Alice is:

$$\mathcal{A}_1 = \{(\text{Sports}), (\text{Comedy})\}$$

The full enumeration of strategies that indicate single actions for Bob is:

$$\mathcal{A}_2 = \{(\text{Sports}, \text{Sports}), (\text{Sports}, \text{Comedy}), (\text{Comedy}, \text{Sports}), (\text{Comedy}, \text{Comedy})\}$$

So $(\text{Sports}, \text{Comedy})$ indicates to choose Sports at B_1 and Comedy at B_2.

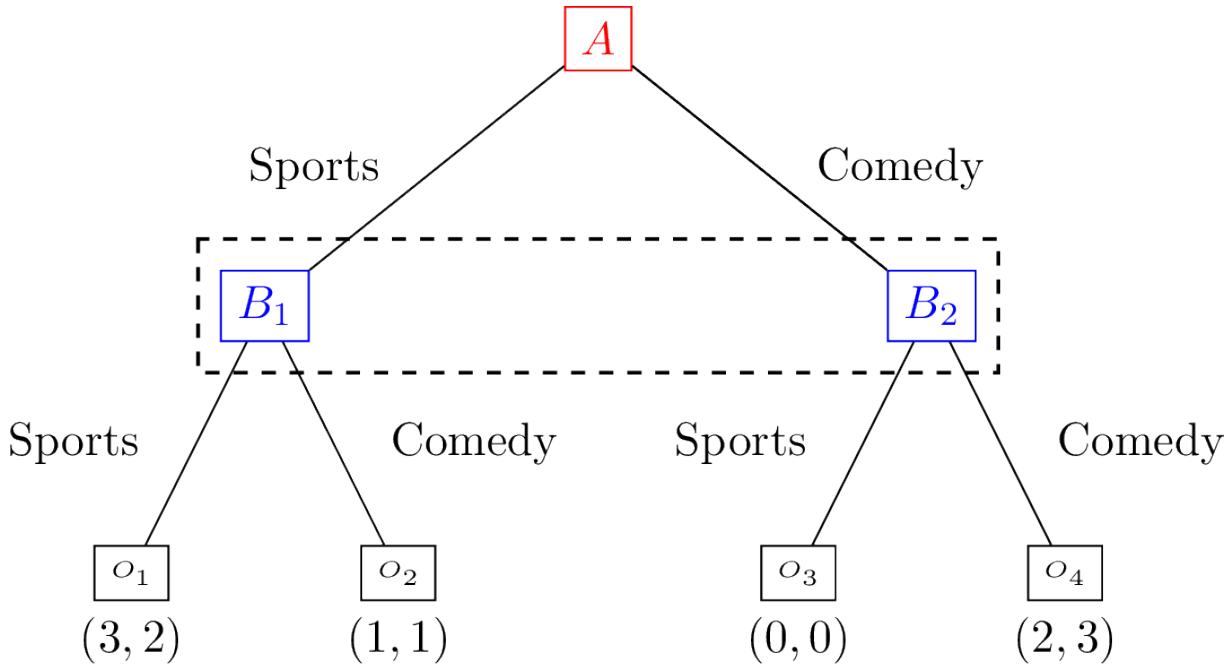
Using this enumeration the payoff functions can be given by the matrices A, B:

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

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

Question

Obtain the Normal Form Game representation corresponding to



Answer

The full enumeration of strategies that indicate single actions for Alice is:

$$\mathcal{A}_1 = \{(\text{Sports}), (\text{Comedy})\}$$

The full enumeration of strategies that indicate single actions for Bob is:

$$\mathcal{A}_2 = \{(\text{Sports}), (\text{Comedy})\}$$

This is because there is a single information set for Bob.

Using this enumeration the payoff functions can be given by the matrices A, B:

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

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

3.6.7 Using Nashpy

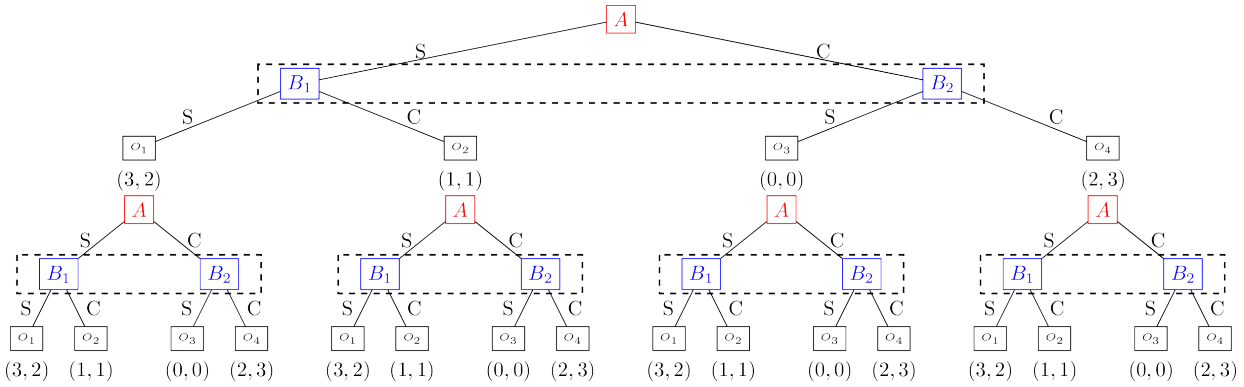
See *Solve with support enumeration* for guidance of how to use Nashpy to use support enumeration to find Nash equilibria once a Normal Form game representation has been obtained.

3.7 Repeated Games

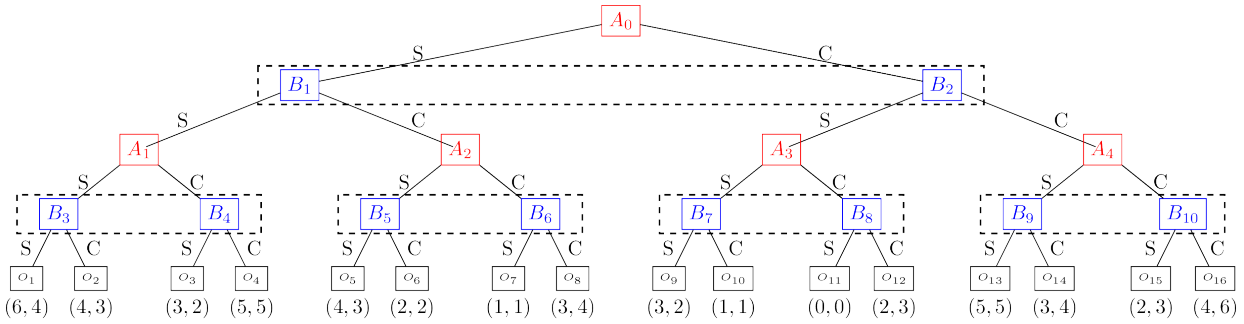
3.7.1 Motivating example: Repeated Coordination Game

Consider the *Coordination game* but in this instance Alice and Bob repeat their play of this game. In other words, they aim to meet (both making their decision at the same time) and after this first meeting they repeat the process, with full knowledge of the outcome of the first play.

This can be represented pictorially as follows:



To show this as an equivalent *extensive form game*, the tree is the same but we take care to label the vertices correctly:



3.7.2 Definition of a repeated games

Given a two player game $(A, B) \in \mathbb{R}^{m \times n^2}$, referred to as a stage game, a T-stage repeated game is a game in which players play that stage game for $T > 0$ repetitions. Players make decisions based on the full history of play over all the repetitions.

Question

For the following values of T and the following stage games, how many leaves would the extensive form representation of the repeated game have:

1.
$$A = \begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix} \qquad B = \begin{pmatrix} 2 & 3 \\ 1 & -1 \end{pmatrix} \qquad T = 2$$

2.
$$A = \begin{pmatrix} 0 & 1 \\ -1 & 3 \end{pmatrix} \qquad B = -A \qquad T = 2$$

3.

$$A = \begin{pmatrix} 0 & 1 \\ -1 & 3 \end{pmatrix} \qquad B = -A \qquad T = 3$$

4.

$$A = \begin{pmatrix} 0 & 1 & 4 \\ 1 & -1 & 3 \end{pmatrix} \qquad B = -A \qquad T = 2$$

Answer

1. The initial play of the game will have 4 leaves (corresponding to the 2 choices by each player), each leave will in turn have 4 leaves. Thus, the total number of leaves will be 16.
2. The initial play of the game will have 4 leaves (corresponding to the 2 choices by each player), each leave will in turn have 4 leaves. Thus, the total number of leaves will be 16.
3. The initial play of the game will have 4 leaves (corresponding to the 2 choices by each player), each leave will in turn have 4 leaves in the second repetition. In the final repetition each of those leaves will have 4 leaves. Thus, the total number of leaves will be 64.
4. The initial play of the game will have 6 leaves (corresponding to the 2 choices by the row player and 3 by the column player), each leave will in turn have 6 leaves in the second repetition. Thus, the total number of leaves will be 36.

3.7.3 Strategies in a repeated game

A strategy for a player in a repeated game is a mapping from all possible histories of play to a probability distribution over the action set of the stage game.

Question

For the *repeated coordination game* which of the following are valid strategies, and in the case of valid strategies what is the outcome.

1. For the row player:

$$\begin{aligned} &(\emptyset, \emptyset) \to C \\ &(S, S) \to C \\ &(S, C) \to C \\ &(C, S) \to S \\ &(C, C) \to S \end{aligned} \tag{3.9}$$

   For the column player:

$$\begin{aligned} &(\emptyset, \emptyset) \to S \\ &(S, S) \to C \\ &(S, C) \to C \\ &(C, S) \to S \\ &(C, C) \to S \end{aligned} \tag{3.15}$$

2. For the row player:

$$(\emptyset, \emptyset) \to C \tag{3.21}$$
$$(S, S) \to C \tag{3.22}$$
$$(C, S) \to C \tag{3.23}$$
$$(C, C) \to C \tag{3.24}$$
$$\tag{3.25}$$

For the column player:

$$(\emptyset, \emptyset) \to S \tag{3.26}$$
$$(S, S) \to C \tag{3.27}$$
$$(S, C) \to S \tag{3.28}$$
$$(C, S) \to S \tag{3.29}$$
$$(C, C) \to S \tag{3.30}$$
$$\tag{3.31}$$

3. For the row player:

$$(\emptyset, \emptyset) \to C \tag{3.32}$$
$$(S, S) \to C \tag{3.33}$$
$$(C, S) \to S \tag{3.34}$$
$$(S, C) \to S \tag{3.35}$$
$$(C, C) \to C \tag{3.36}$$
$$\tag{3.37}$$

For the column player:

$$(\emptyset, \emptyset) \to S \tag{3.38}$$
$$(S, S) \to C \tag{3.39}$$
$$(S, C) \to C \tag{3.40}$$
$$(C, S) \to S \tag{3.41}$$
$$(C, C) \to S \tag{3.42}$$
$$\tag{3.43}$$

4. For the row player:

$$(\emptyset, \emptyset) \to S \tag{3.44}$$
$$(S, S) \to C \tag{3.45}$$
$$(C, S) \to S \tag{3.46}$$
$$(S, C) \to C \tag{3.47}$$
$$(C, C) \to S \tag{3.48}$$
$$\tag{3.49}$$

For the column player:

$$(\emptyset, \emptyset) \to S \tag{3.50}$$
$$(S, S) \to C \tag{3.51}$$
$$(S, C) \to C \tag{3.52}$$
$$(C, S) \to S \tag{3.53}$$
$$(C, C) \to S \tag{3.54}$$
$$\tag{3.55}$$

Answer

1. This is a valid strategy pair: all possible histories are mapped to correct actions. The outcome would be: $(3, 2)$ (corresponding to O_9 of the extensive form representation).
2. This is not a valid strategy pair: the row player strategy does not have a mapping from (S, C).
3. This is not a valid strategy pair: the column player strategy maps from (C, S) to an action (α) that is not in the action space of the stage game.
4. This is a valid strategy pair: all possible histories are mapped to correct actions. The outcome would be: $(5, 5)$ (corresponding to O_4 of the extensive form representation).

3.7.4 Equilibria in repeated games

In a repeated game it is possible for players to encode reputation and trust in their strategies.

Consider as an example the following stage game with $T = 2$:

$$A = \begin{pmatrix} 0 & 6 & 1 \\ 1 & 7 & 5 \end{pmatrix} \qquad B = \begin{pmatrix} 0 & 3 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

Through inspection it is possible to verify that the following strategy pair is a Nash equilibrium:

For the row player:

$$\begin{aligned} (\emptyset, \emptyset) &\to r_1 \\ (r_1, c_1) &\to r_2 \\ (r_1, c_2) &\to r_2 \\ (r_1, c_3) &\to r_2 \\ (r_2, c_1) &\to r_2 \\ (r_2, c_2) &\to r_2 \\ (r_2, c_3) &\to r_2 \end{aligned} \tag{3.56}$$

(3.63)

For the column player:

$$\begin{aligned} (\emptyset, \emptyset) &\to c_2 \\ (r_1, c_1) &\to c_3 \\ (r_2, c_1) &\to c_1 \\ (r_1, c_2) &\to c_3 \\ (r_2, c_2) &\to c_1 \\ (r_1, c_3) &\to c_3 \\ (r_2, c_3) &\to c_1 \end{aligned} \tag{3.64}$$

(3.71)

This pair of strategies correspond to the following scenario:

The row player plays r_1 and the column player plays c_2 in the first state. The row player plays r_2 and the column player plays c_3 in the second stage.

Note that if the row player deviates and plays r_2 in the first stage then the column player will play c_1.

If both players play these strategies their utilities are: $(11, 4)$ which is better **for both players** then the utilities at any sequence of pure stage Nash equilibria. **But** is this a Nash equilibrium? To find out we investigate if either player has an incentive to deviate.

1. If the row player deviates, they would only be rational to do so in the first stage, if they did they would gain 1 in that stage but lose 4 in the second stage. Thus they have no incentive to deviate.
2. If the column player deviates, they would only do so in the first stage and gain no utility.

Thus this strategy pair **is a Nash equilibrium** and evidences how a reputation can be built and cooperation can emerge from complex dynamics.

3.7.5 Using Nashpy

Repeated games are a particularly compact way of representing a given subset of *Extensive Form Games*. Thus, it is possible to study them as an equivalent *normal form game*. See *Obtain a repeated game* for guidance of how to use Nashpy to generate a normal form game by repeating a stage game.

3.8 The Emergence of Cooperation

3.8.1 Motivating example: can cooperation emerge in a short time frame?

The repeated *Prisoners Dilemma* is a model of direct reciprocity.

$$A = \begin{pmatrix} 3 & 0 \\ 5 & 1 \end{pmatrix}$$

1. The first action corresponds to acting in the interest of all individuals (this is often referred to as Cooperate).
2. The second action corresponds to acting in ones own self interest (this is often referred to as Defect).

In a population of individuals, is it possible for self interest to not become the norm?

To answer this question we can consider the Iterated Prisoners Dilemma as a *repeated game* where individuals have **two** consecutive interactions. The incentive to initially acting selflessly is that the other individual will do the same in the second interaction.

Recalling *Strategies in a repeated game* the strategies in this case must map histories of play to actions. The possible histories of play are:

$$\mathcal{H} = \{(\emptyset, \emptyset), (r_1, c_1), (r_1, c_2), (r_2, c_1), (r_2, c_2)\}$$

Row strategies are thus of the form:

Cooperate unconditionally:

$$\begin{aligned} (\emptyset, \emptyset) &\to r_1 \\ (r_1, c_1) &\to r_1 \\ (r_1, c_2) &\to r_1 \\ (r_2, c_1) &\to r_1 \\ (r_2, c_2) &\to r_1 \end{aligned} \tag{3.72}$$

(3.77)

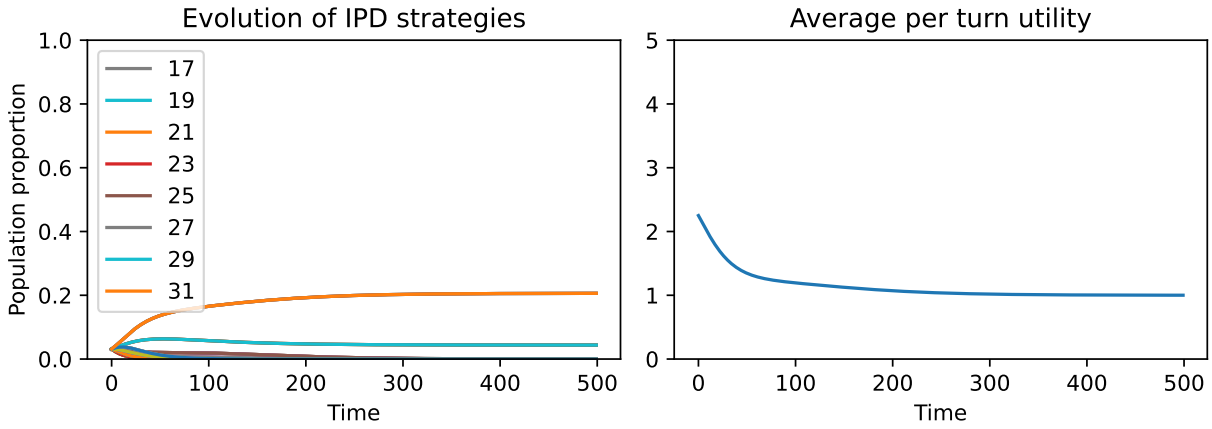
Defect unconditionally:

$$(\emptyset, \emptyset) \to r_2 \tag{3.78}$$
$$(r_1, c_1) \to r_2 \tag{3.79}$$
$$(r_1, c_2) \to r_2 \tag{3.80}$$
$$(r_2, c_1) \to r_2 \tag{3.81}$$
$$(r_2, c_2) \to r_2 \tag{3.82}$$
$$\tag{3.83}$$

Start by cooperating and then repeat the action of the opponent:

$$(\emptyset, \emptyset) \to r_1 \tag{3.84}$$
$$(r_1, c_1) \to r_1 \tag{3.85}$$
$$(r_1, c_2) \to r_2 \tag{3.86}$$
$$(r_2, c_1) \to r_1 \tag{3.87}$$
$$(r_2, c_2) \to r_2 \tag{3.88}$$
$$\tag{3.89}$$

The strategy space when repeating the game **twice** corresponds to 32 different strategies. We can see how these 32 strategies interact in an evolutionary setting using *replicator dynamics*.



The legend shows the index in the strategy space of the strategies that have a final proportion larger than 10^{-2}. The average utility plot gives us the answer to our question: the average per turn utility is 1 which implies that the strategies that survive the evolutionary process are the ones that act selfishly.

The immediate conclusion is somewhat disappointing: how can a society emerge in which individuals will do what is best for the collective?

This question can be better answered by considering a much larger strategy space corresponding to more repetitions of the prisoners dilemma.

3.8.2 The General form of the Prisoners Dilemma

The general form is:

$$A = \begin{pmatrix} R & S \\ T & P \end{pmatrix} \qquad B = \begin{pmatrix} R & T \\ S & P \end{pmatrix}$$

with the following constraints:

$$T > R > P > S \qquad 2R > T + S$$

- The first constraint ensures that the second action "Defect" dominates the first action "Cooperate".
- The second constraint ensures that a social dilemma arises: the sum of the utilities to both players is best when they both cooperate.

This game is a good model of agent (human, etc) interaction: a player can choose to take a slight loss of utility for the benefit of the other play **and** themselves.

Question

Under what conditions is the following game a Prisoners Dilemma:

$$A = \begin{pmatrix} 1 & -\mu \\ 1 + \mu & 0 \end{pmatrix} \qquad B = A^T$$

Answer

This is a Prisoners Dilemma when:

$$1 + \mu > 10 > -\mu$$

and:

$$2 > 1$$

Both of these equations hold for $\mu > 0$. This is a convenient form for the Prisoners Dilemma as it corresponds to a 1 dimensional parametrization.

As a single one shot game there is not much more to say about the Prisoner's dilemma. It becomes fascinating when studied as a repeated game.

3.8.3 Axelrod's tournaments

In 1980, Robert Axelrod (a political scientist) invited submissions to a computer tournament version of an iterated prisoners dilemma. This was described in a 1980 paper [Axelrod1980].

- 14 strategies were submitted.
- Round robin tournament with 200 stages including a 15th player who played uniformly randomly.
- Some complicated strategies, including for example a strategy that used a χ^2 test to try and identify strategies that were acting randomly. You can read more about this tournament here: http://axelrod.readthedocs.io/en/stable/reference/overview_of_strategies.html#axelrod-s-first-tournament

- The winner (average score) was in fact a straightforward strategy: Tit For Tat. This strategy starts by cooperating and then repeats the opponents previous move.

The 15 by 15 payoff matrix (rounded to 1 digit) that corresponds to this tournament is:

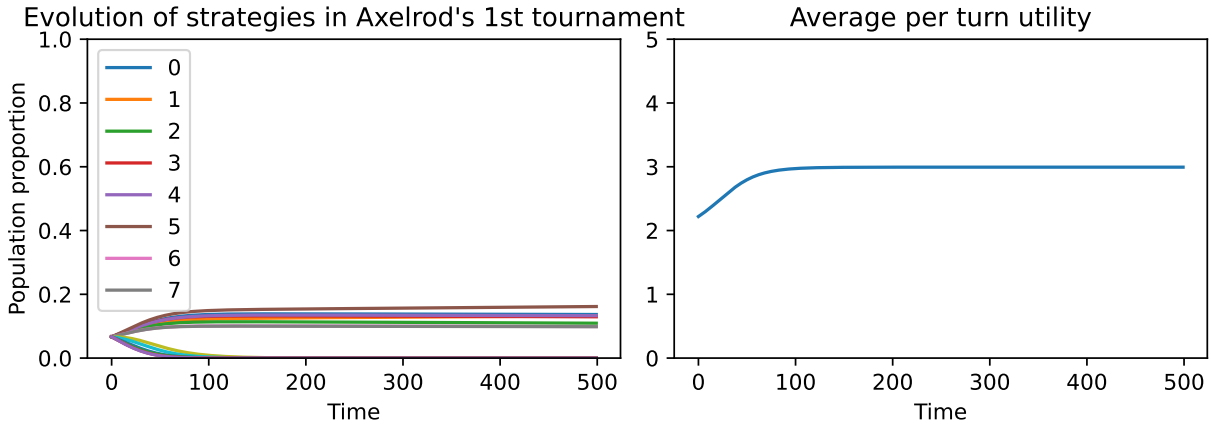
$$\begin{pmatrix}
3. & 3. & 3. & 3. & 3. & 3. & 3. & 3. & 2.6 & 3. \\
1.4 & 1.1 & 1.5 & 2.2 & 2.2 & & & & & \\
3. & 3. & 3. & 3. & 3. & 3. & 3. & 3. & 3. & 1.2 \\
1.3 & 1.1 & 1.3 & 2.8 & 2.8 & & & & & \\
3. & 3. & 3. & 3. & 3. & 3. & 3. & 3. & 2.8 & 0.1 \\
2.2 & 2.7 & 2.7 & 1.6 & 1.5 & & & & & \\
3. & 3. & 3. & 3. & 3. & 3. & 3. & 3. & 3.1 & 0.5 \\
2.1 & 2.4 & 2.4 & 2. & 2.1 & & & & & \\
3. & 3. & 3. & 3. & 3. & 3. & 3. & 3. & 3.3 & 1.3 \\
1.3 & 1.3 & 1.4 & 2.8 & 2.6 & & & & & \\
3. & 3. & 3. & 3. & 3. & 3. & 3. & 3. & 2.6 & 2.5 \\
1.4 & 1.2 & 1.7 & 2.9 & 2.9 & & & & & \\
3. & 3. & 3. & 3. & 3. & 3. & 3. & 3. & 2. & 1.1 \\
1.2 & 1.1 & 1.3 & 3. & 3. & & & & & \\
3. & 3. & 3. & 3. & 3. & 3. & 3. & 3. & 2. & 1. \\
1.3 & 1.1 & 1.2 & 3. & 2.9 & & & & & \\
2.6 & 2.8 & 3.2 & 2.7 & 1.8 & 2.6 & 1.4 & 1.4 & 1.5 & 3.1 \\
1.5 & 1.4 & 2.1 & 2.7 & 2.9 & & & & & \\
3. & 1. & 4.9 & 3.3 & 1.4 & 1.1 & 1. & 1.2 & 2.7 & 1. \\
2.2 & 2.6 & 1.2 & 2.7 & 2.6 & & & & & \\
1.4 & 1.3 & 3.5 & 2.8 & 1.5 & 1.4 & 1.2 & 1.3 & 1.7 & 3.5 \\
1.3 & 1.1 & 1.3 & 2.4 & 2.4 & & & & & \\
1.2 & 1.1 & 3.2 & 2.8 & 1.4 & 1.2 & 1.1 & 1.1 & 1.5 & 3.2 \\
1.2 & 1.2 & 1.3 & 2.3 & 2.3 & & & & & \\
1.5 & 1.3 & 3.2 & 2.8 & 1.6 & 1.7 & 1.2 & 1.1 & 2.4 & 1. \\
1.3 & 1.2 & 1.4 & 2.3 & 2.4 & & & & & \\
2.2 & 0.9 & 3.9 & 2.8 & 1.1 & 0.8 & 0.6 & 0.6 & 1.2 & 1.2 \\
1.8 & 2.1 & 2. & 2.3 & 2.3 & & & & & \\
2.3 & 0.9 & 3.9 & 2.7 & 1.2 & 0.7 & 0.5 & 0.7 & 1. & 1.2 \\
1.8 & 2.1 & 2. & 2.3 & 2.3 & & & & &
\end{pmatrix}$$

We see that the first 8 strategies all cooperate with each other (getting a utility of 3).

These 15 strategies are a small subset of the strategy space for the iterated prisoners dilemma with $T = 200$ repetitions. As before, we can see how these 15 strategies interact in an evolutionary setting using *replicator dynamics*.

It is evident here that cooperation emerges from this strategy space.

The fact that Tit For Tat won garnered a lot of research (still ongoing) as it showed a mathematical model of how cooperative behaviour can emerge in complex situations. However, recent research has shown that Tit For Tat is not a universally strong strategy [Knight2018], [Harper2017], [Press2012].



3.8.4 Using Python

There is a Python library (`axelrod`) with over 200 strategies that can be used to reproduce this work [Knight2016]. You can read the documentation for it here: http://axelrod.readthedocs.io.

3.9 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.

3.9.1 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 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:

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

3.10 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 run with two optional arguments:

- `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 make 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 strict run 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():
...     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]
```

3.11 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 interval 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.

3.11.1 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)):
```

(continues on next page)

(continued from previous page)

```
...     plt.plot(strategy, label=f"$s_{number}$")
>>> plt.xlabel("Iteration")
>>> plt.ylabel("Probability")
>>> plt.title("Actions taken by row player")
>>> plt.legend()
```

3.12 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 interval 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:

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

3.12.1 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
```

(continues on next page)

(continued from previous page)

```
>>> 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()
```

3.13 Replicator dynamics

3.13.1 Motivating example: The Hawk Dove Game

Consider a population of animals. These animals when they interact will always share their food. Due to a genetic mutation, some of these animals may act in an aggressive manner and not share their food. If two aggressive animals meet they both compete and end up with no food. If an aggressive animal meets a sharing one, the aggressive one will take most of the food.

These interactions can be represented using the matrix A:

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

In this scenario: what is the likely long term effect of the genetic mutation?

Over time will:

- The population resist the mutation and all the animals continue to share their food.
- The population get taken over by the mutation and all animals become aggressive.
- A mix of animals are present in the population some act aggressively and some share.

To answer this question we will assume a vector x represents the population. In this case:

- x_1 represents the proportion of the population that shares.
- x_2 represents the proportion of the population that acts aggressively.

Note that as the components of x are proportions of the population this implies:

$$\sum_i x_i = 1$$

We will also assume that any given individual in the population is playing a *strategy* χ, which:

- shares χ_1 proportion of the time.
- acts aggressively χ_2 proportion of the time.

The overall fitness of an individual in the population is then given by their expected utility (as given by A) as they interact with the population:

$$\chi A x$$

We can in fact write down the fitness corresponding to each action (sharing or being aggressive):

$$f = Ax$$

The average fitness in the population is then given by:

$$\phi = x^T f$$

To understand how the population will evolve relative to their fitness the following differential equation will be used:

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

In our case the differential equations are:

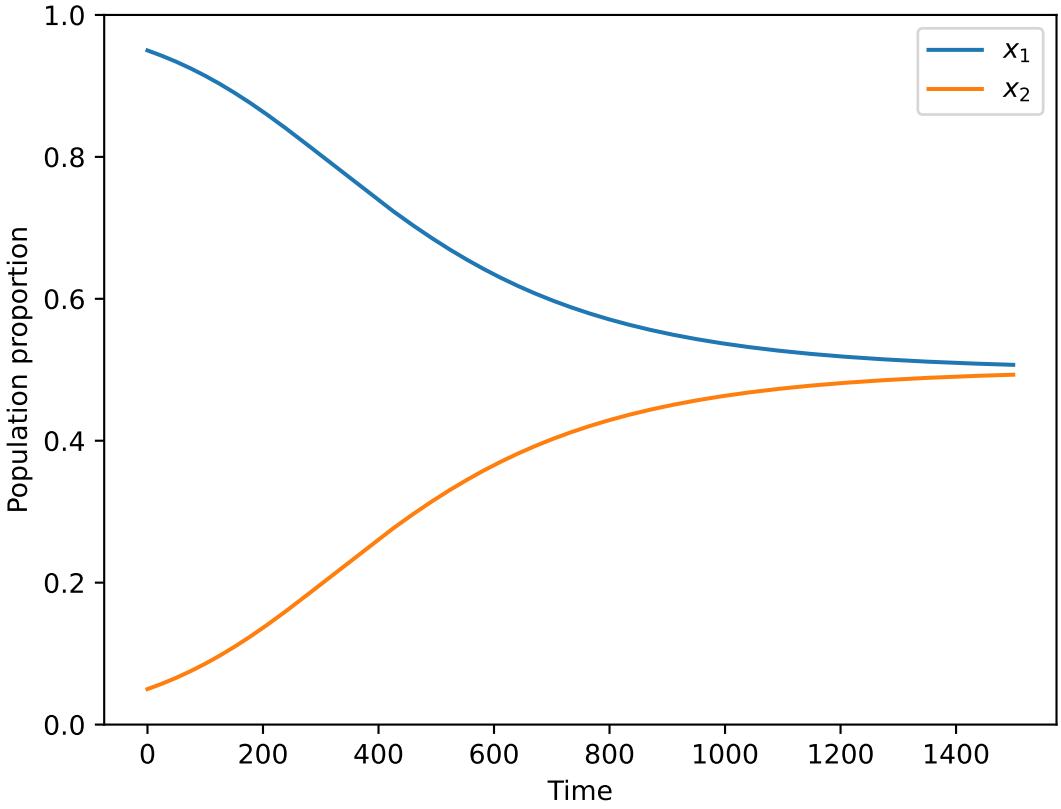
$$\frac{dx_1}{dt} = x_1(2x_1 + x_2 - \phi) \tag{3.90}$$

$$\frac{dx_2}{dt} = x_2(3x_1 - \phi) \tag{3.91}$$

where:

$$\phi = x_1(2x_1 + x_2) + x_2(3x_1)$$

This differential equation can then be *solved numerically* to show the evolution of the population over time. We can see that it looks like in our particular situation the mutation stays within the population and a mix of both sharing and aggressive animals will coexist.



3.13.2 The replicator dynamics equation

Given a population with N types of individuals. Where the fitness of an individual of type i when interacting with an individual of type j is given by A_{ij} where $A \in \mathbb{R}^{N \times N}$. The replicator dynamics equation is given by:

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

where:

$$\phi = \sum_{i=1}^{N} x_i f_i(x)$$

where f_i is the population dependent fitness of individuals of type i:

$$f_i(x) = (Ax)_i$$

Note that there are equivalent linear algebraic definitions to the above:

$$f = Ax \qquad \phi = x^T Ax$$

Question

For *Rock Paper Scissors*, what is the replicator dynamics equation?

Answer

Recalling that rock paper scissors has a payoff matrix A given by:

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

For a general population vector x the population dependent fitness f is given by:

$$f = Ax = \begin{pmatrix} -x_2 + x_3 \\ x_1 - x_3 \\ -x_1 + x_2 \end{pmatrix}$$

The average fitness is given by:

$$\phi = x^T f = x_1(x_3 - x_2) + x_2(x_1 - x_3) + x_3(x_2 - x_1)$$

The replicator dynamics equation is then given by:

$$\frac{dx_1}{dt} = x_1(x_3 - x_2 - \phi) \tag{3.92}$$
$$\frac{dx_2}{dt} = x_2(x_1 - x_3 - \phi) \tag{3.93}$$
$$\frac{dx_3}{dt} = x_3(x_2 - x_1 - \phi) \tag{3.94}$$

Closer inspection of ϕ gives: $\phi = 0$ thus:

$$\frac{dx_1}{dt} = x_1(x_3 - x_2) \tag{3.95}$$
$$\frac{dx_2}{dt} = x_2(x_1 - x_3) \tag{3.96}$$
$$\frac{dx_3}{dt} = x_3(x_2 - x_1) \tag{3.97}$$

3.13.3 Stability of the replicator dynamics equation

Stability of the replicator dynamics equation is achieved when $\frac{dx_i}{dt} = 0$ for all i.

For a population vector x^* for which $\frac{dx_i^*}{dt} = 0$ for all i the population will not change without some other effect. This is referred to as a **stable population**.

Question

For the following games, what are the stable populations?

1. *Rock Paper Scissors*

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

2. *Hawk Dove Game*

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

Answer

1. The replicator dynamics equation for this game are:

$$\frac{dx_1}{dt} = x_1(x_3 - x_2) \tag{3.98}$$
$$\frac{dx_2}{dt} = x_2(x_1 - x_3) \tag{3.99}$$
$$\frac{dx_3}{dt} = x_3(x_2 - x_1) \tag{3.100}$$

For them all to be 0, this requires:

- $x_1 = 0$ or $x_2 = x_3$
- $x_2 = 0$ or $x_1 = x_3$
- $x_3 = 0$ or $x_1 = x_2$

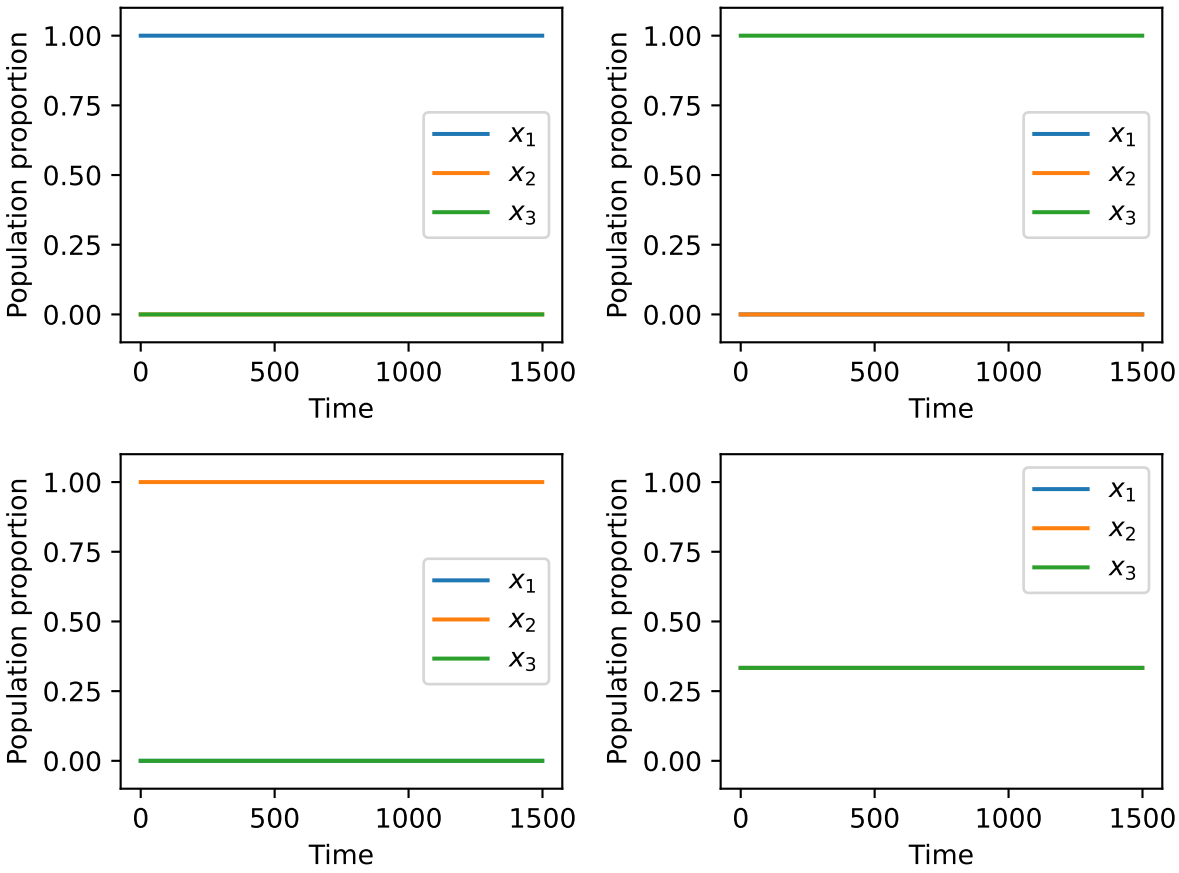
Which, through inspection in turn requires:

- $x_1 \neq 0$ and $x_2 = x_3 = 0$ or
- $x_2 \neq 0$ and $x_1 = x_3 = 0$ or
- $x_3 \neq 0$ and $x_1 = x_2 = 0$ or
- $x_1 = x_2 = x_3$.

Given that $x_1 + x_2 + x_3 = 1$ this leaves us with 4 possible stable populations:

1. $x = (1, 0, 0)$
2. $x = (0, 1, 0)$
3. $x = (0, 0, 1)$
4. $x = (1/3, 1/3, 1/3)$

The following plot shows each of the above populations which no longer change over time:



2. The replicator dynamics equation for this game are:

$$\frac{dx_1}{dt} = x_1(2x_1 + x_2 - \phi) \tag{3.101}$$

$$\frac{dx_2}{dt} = x_2(3x_1 - \phi) \tag{3.102}$$

where:

$$\phi = x_1(2x_1 + x_2) + x_2(3x_1)$$

substituting $x_2 = 1 - x_1$ here gives:

$$\frac{dx_1}{dt} = x_1(x_1 - 1)(2x_1 - 1) \tag{3.103}$$

$$\frac{dx_2}{dt} = -x_1(x_1 - 1)(2x_1 - 1) \tag{3.104}$$

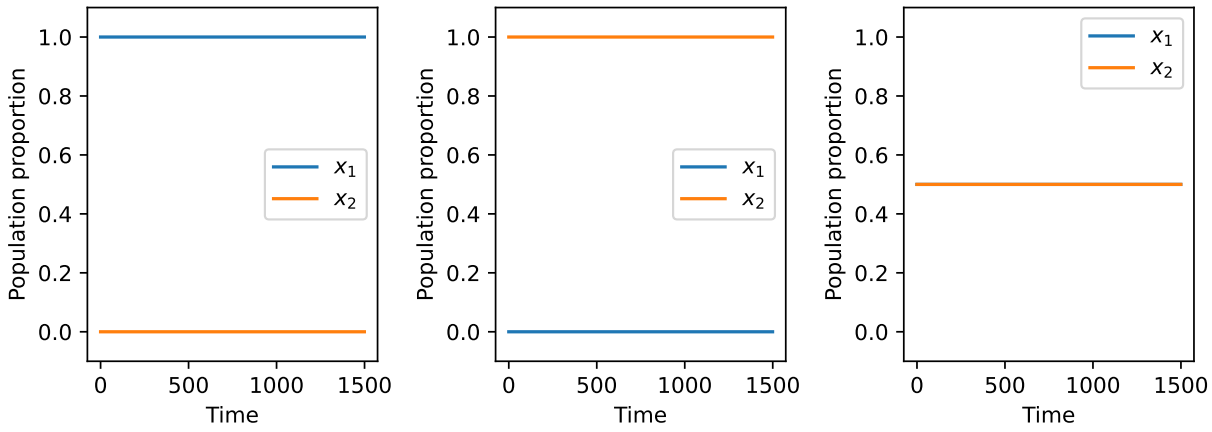
For them both to be 0, this requires:

- $x_1 = 0$ or
- $x_1 = 1$ or
- $x_1 = 1/2$

Recalling the substition that $x_2 = 1 - x_1$ this leaves us with 3 possible stable populations:

1. $x = (1, 0)$
2. $x = (0, 1)$
3. $x = (1/2, 1/2)$

The following plot shows each of the above populations which no longer change over time:



3.13.4 Evolutionary stable strategies

Evolutionary stable strategies are strategies that when adopted by an entire population are resistant to an alternative strategy that is initially rare.

By definition an evolutionary stable strategy corresponds to a stable population.

For the *hawk dove game* there are 3 stable populations:

- $x = (1, 0)$
- $x = (0, 1)$
- $x = (1/2, 1/2)$

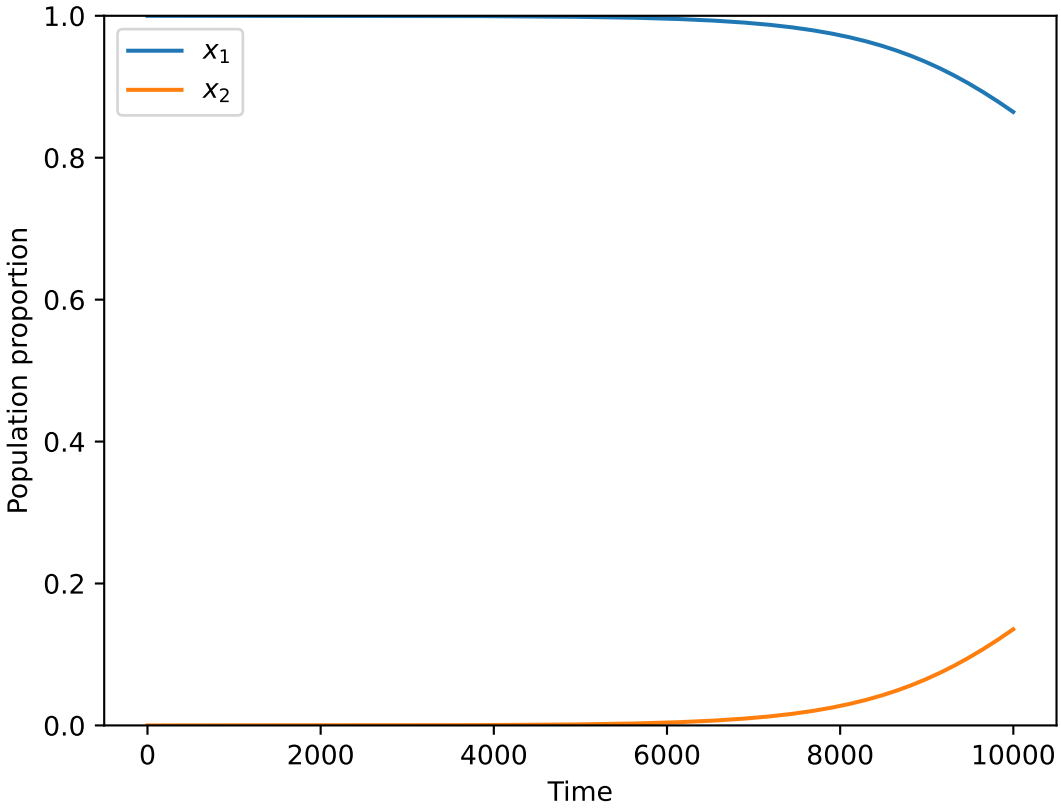
However, if a small deviation is made from the first two populations then the population does not "resist". For example, we consider the initial population $x = (1, 0)$ and introduce a small population aggressive behaviours to have: $x = (1 - \epsilon, \epsilon)$ where $\epsilon > 0$. The plot below shows this with $\epsilon = 10^-5$:

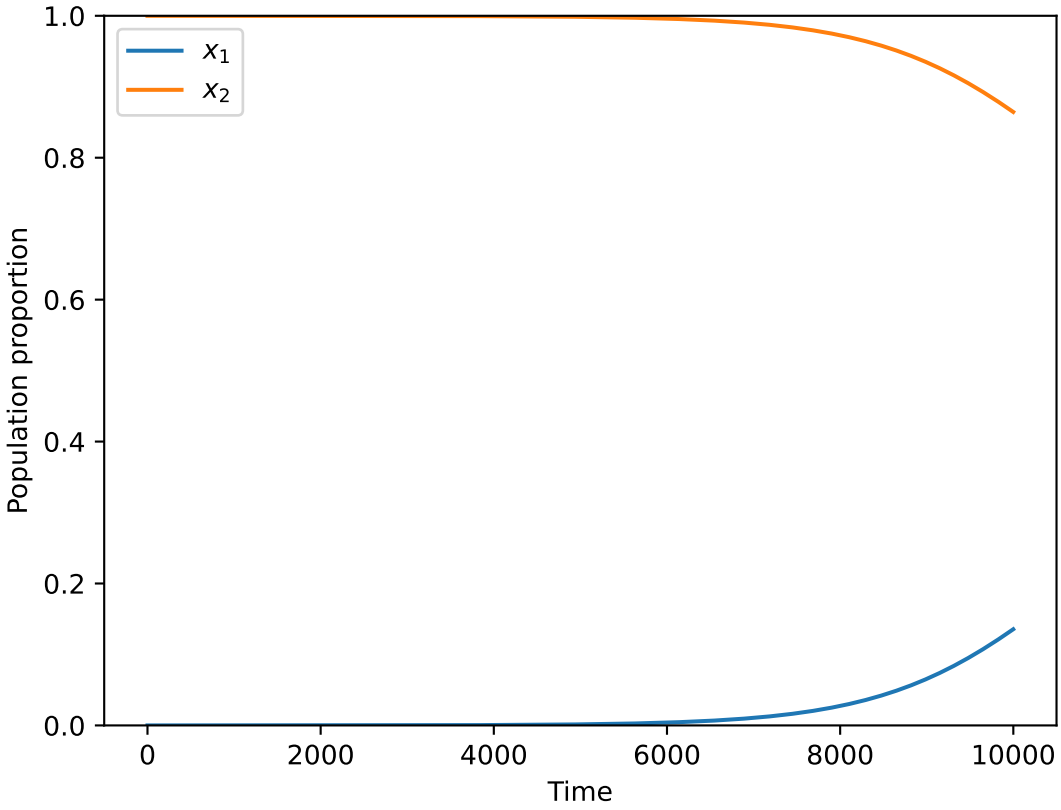
This is also what happens if we start with a population of aggressive animals: We consider the initial population $x = (0, 1)$ and introduce a small population aggressive behaviours to have: $x = (\epsilon, 1 - \epsilon)$ where $\epsilon > 0$. The plot below shows this with $\epsilon = 10^{-5}$:

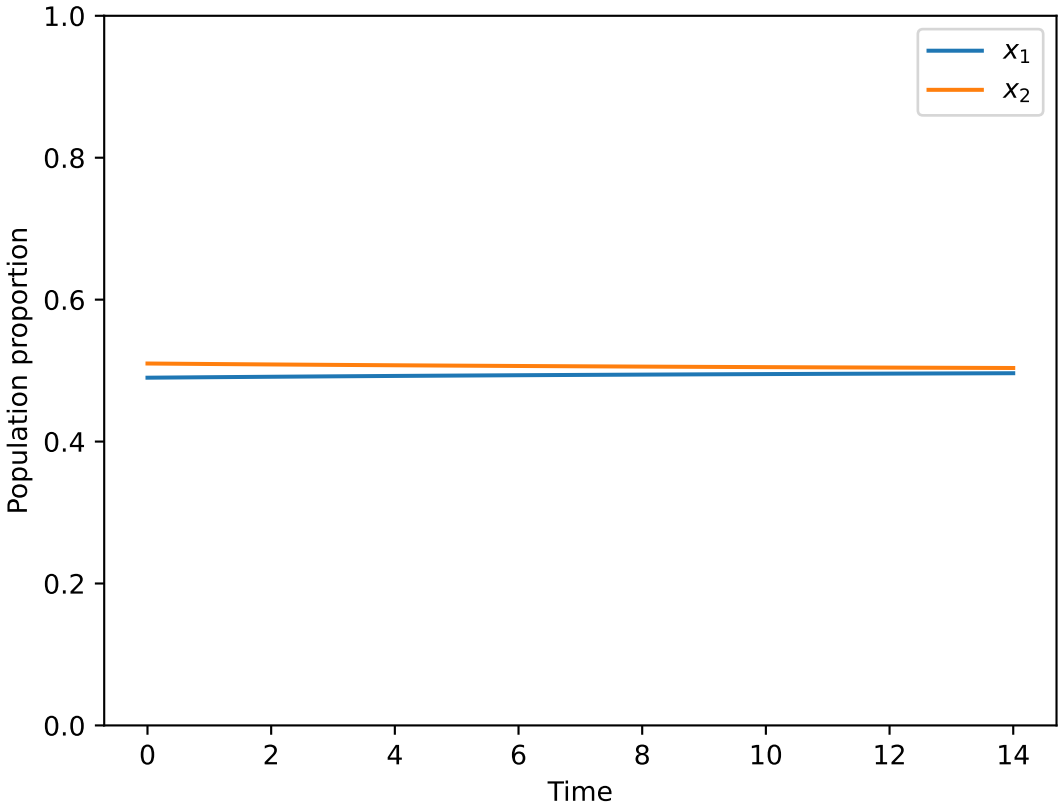
However, this is not the case with the third stable population: $x = (1/2, 1/2)$. The plot below shows $x = (1/2 - \epsilon, 1/2 + \epsilon)$ with $\epsilon = 10^{-2}$:

These observations can be confirmed analytically. Information on this can be found in [Fudenberg1998], [Webb2007] and [Nowak2006].

The replicator equations were first presented in [Maynard1974].


1.0
0.8
0.6
0.4
0.2
0.0
Population proportion
x_1
x_2
0
2000
4000
6000
8000
10000
Time


1.0
0.8
0.6
0.4
0.2
0.0
Population proportion
x_1
x_2
0
2000
4000
6000
8000
10000
Time


1.0
0.8
0.6
0.4
0.2
0.0
Population proportion
0
2
4
6
8
10
12
14
Time
x_1
x_2

3.13.5 The replicator-mutation dynamics equation

An extension of the *replicator equation* is to allow for mutation [Komarova2004]. In this case reproduction is imperfect and individuals of a given type can give individuals of another.

This is expressed using a matrix Q where Q_{ij} denotes the probability of an individual of type j is produced by an individual of type i.

In this case the replicator equation can be modified to give the replicator-mutation equation:

$$\frac{dx_i}{dt} = \sum_{j=1}^{N} x_j f_j Q_{ji} - x_i \phi \text{ for all } i$$

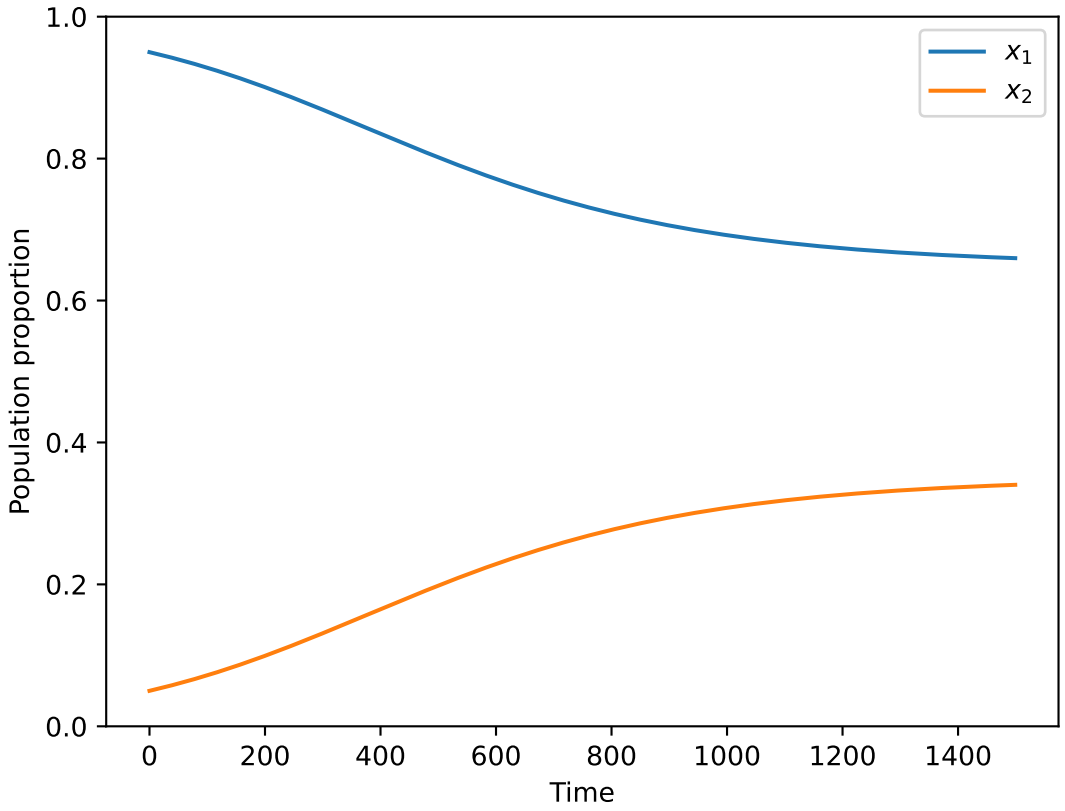
where, as before:

$$f = Ax \qquad \phi = x^T Ax$$

This can modify emergent behaviour. For the *Hawk Dove game* if there is a 10% change that aggressive individuals will produce sharing ones the matrix Q is given by:

$$Q = \begin{pmatrix} 1 & 0 \\ 1/10 & 9/10 \end{pmatrix}$$

The plot below shows the evolution of the system:



Question

Show that for $Q = I_N$ (the identity matrix of size N) the replicator-mutation equation corresponds to the replicator equation.

Answer

The replicator-mutation equation is:

$$\frac{dx_i}{dt} = \sum_{j=1}^{N} x_j f_j Q_{ji} - x_i \phi \text{ for all } i$$

As $Q = I_N$:

$$Q_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

This gives:

$$\frac{dx_i}{dt} = x_i f_i Q_{ii} - x_i \phi \text{ for all } i \qquad Q_{ij} = 0 \text{ for all } i \neq j \tag{3.105}$$

$$\frac{dx_i}{dt} = x_i f_i - x_i \phi \text{ for all } i \tag{3.106}$$

$$\frac{dx_i}{dt} = x_i (f_i - \phi) \text{ for all } i \tag{3.107}$$

As required.

3.13.6 Using Nashpy

See *Use replicator dynamics* for guidance of how to use Nashpy to obtain numerical solutions of the replicator dynamics equation. See *Use replicator dynamics with mutation* for guidance of how to use Nashpy to obtain numerical solutions of the replicator-mutation dynamics equation.This is what is used to obtain all the plots above.

3.14 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:

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

3.14.1 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"$s_{1}$"])
```

3.15 Moran Processes

3.15.1 Motivating example: The Hawk Dove Game

Consider a **finite** population of N animals. Similarly to *the motivating example for replicator dynamics*, these animals when they interact will always share their food. Due to a genetic mutation, some of these animals may act in an aggressive manner and not share their food. If two aggressive animals meet they both compete and end up with no food. If an aggressive animal meets a sharing one, the aggressive one will take most of the food.

The difference with a replicator dynamics model is that it will be assumed that the size of the population is finite and stays constant.

These interactions can be represented using the matrix A:

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

In this scenario: what is the probability that the mutation takes over the entire population?

To answer this question we will assume a vector v represents the population. In this case:

- v_1 represents the number of individuals of the population that share.
- v_2 represents the number of individuals of the population that act aggressively.

Note that as the size of the population is assumed to be constant this implies that:

$$\sum_i x_i = N$$

Where N is the number of individuals in the population.

The overall fitness of an individual of a given type in a population v is then given by the expected utility (as given by A) of individuals of that type as they interact with the population:

$$f_1 = \frac{2(v_1 - 1) + 1v_2}{N - 1} \tag{3.108}$$

$$f_2 = \frac{3v_1 + 1(v_2 - 1)}{N - 1} \tag{3.109}$$

Note that f_i is dependent on v.

The evolutionary process defined in this chapter will assume an individual will be selected for copy proportional to their fitness. The probability of picking an individual of a given type i is thus given by:

$$\frac{v_i f_i}{v_1 f_1 + v_2 f_2}$$

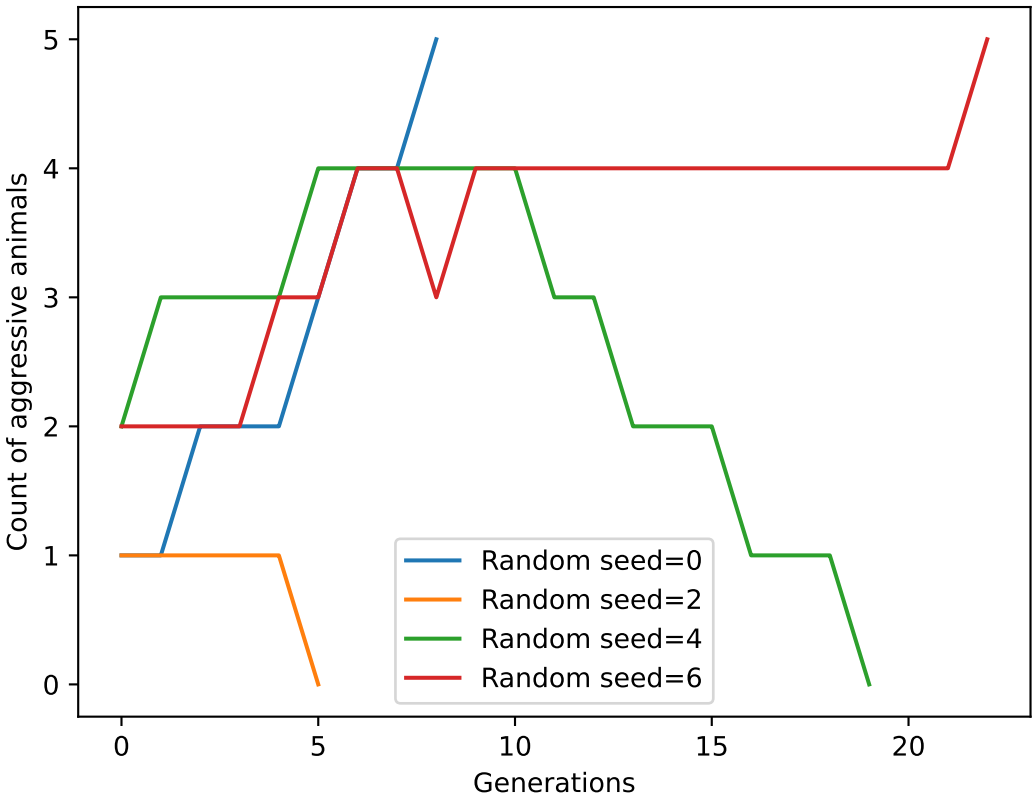
To ensure the population stays constant this requires that an individual is chosen to be removed. This is done uniformly randomly. The probability of picking an individual of a given type i for removal is then given by:

$$\frac{v_i}{v_1 + v_2}$$

These probabilities allow us to define a Markov process that describes the evolution of the system. Here is a diagram showing the different states in the process for $N = 5$.

The answer to our question corresponds to the probability that starting in state $v = (4, 1)$, we arrive at state $v = (0, 5)$.

The following plot shows 4 possible outcomes. In 2 of them the sharing animals resist the invasion of the aggressive one.



3.15.2 The Moran process

First defined in [Moran1958] the Moran process assumes a constant population of N individuals which can be of m different types. There exists a fitness function $f : [1, \ldots, m] \times [1, \ldots, m]^N \to \mathbb{R}$ that maps each individual to a numeric fitness value which is dependent on the types of the individuals in the population.

The process is defined as follows, at each step:

1. Every individual k has their fitness f_k calculated.
2. An individual is randomly selected for copying. This selection is done proportional to their fitness: $f_k(v)$. Thus, the probability of selecting individual k for copying is given by:

$$\frac{f_k(v)}{\sum_{h=1^N} f_h(v)}$$

3. An individual is selected for removal. This selection is done uniformly randomly. Thus, the probability of selecting individual i for removal is given by:

$$1/N$$

4. An individual of the same type as the individual selected for copying is introduced to the population.
5. The individual selected for removal is removed.

The process is repeated until there is only one type of individual left in the population.

Fitness function on a game

A common representation of the fitness function f is to use a game.

As an example consider a population with $N = 10$ and $m = 3$ types of individuals. The fitness of a given individual is calculated by considering the utilities received by each individual when they interact with all other individuals. These interactions are given by the 3×3 matrix A:

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

A_{ij} represents the utility of an individual of type i interacting with an individual of type j.

In this setting, the fitness of an individual of type i is:

$$f_i(v) = (v_i - 1)A_{ii} + \sum_{j \neq i, j=1}^{N} v_j A_{ij}$$

For example, if $v = (4, 5, 1)$ then the fitness of individuals of each type are given by:

1. Individuals of the first type:

$$3 \times 3 + 5 \times 2 + 1 \times 1 = 20$$

2. Individuals of the second type:

$$4 \times 3 + 4 \times 1 + 1 \times 2 = 18$$

3. Individuals of the third type:

$$0 \times 3 + 4 \times 2 + 5 \times 1 = 13$$

Selection probabilities on a game

The probability of selecting an individual of type i for copying is given by:

$$\frac{v_i \times \left((v_i - 1)A_{ii} + \sum_{j \neq i, j=1}^{N} v_j A_{ij}\right)}{\sum_{i=1}^{m} v_i \times \left((v_i - 1)A_{ii} + \sum_{j \neq i, j=1}^{N} v_j A_{ij}\right)}$$

So for this 3×3 example, the probability of selecting an individual of each type for copying is given by:

1. Individuals of the first type:

$$\frac{4 \times 20}{4 \times 20 + 5 \times 18 + 1 \times 13} = \frac{80}{183}$$

2. Individuals of the second type:

$$\frac{5 \times 18}{4 \times 20 + 5 \times 18 + 1 \times 13} = \frac{90}{183}$$

3. Individuals of the third type:

$$\frac{1 \times 13}{4 \times 20 + 5 \times 18 + 1 \times 13} = \frac{13}{183}$$

Question

For the *hawk dove game* what are the probabilities of selecting an individual for copying and for removal for the following populations:

1. $v = (4, 5)$
2. $v = (3, 0)$
3. $v = (6, 6)$

Answer

1. For $v = (4, 5)$ the probabilities are given by:

Type	Copying	Removal
Sharing	$\frac{4(3\times2+5\times1)}{4(3\times2+5\times1)+5(4\times3+4\times0)} = \frac{44}{104}$	$4/9$
Aggresive	$\frac{5(4\times3+4\times0)}{4(3\times2+5\times1)+5(4\times3+4\times0)} = \frac{60}{104}$	$5/9$

2. For $v = (3, 0)$ the probabilities are given by:

Type	Copying	Removal
Sharing	1	1
Aggresive	0	0

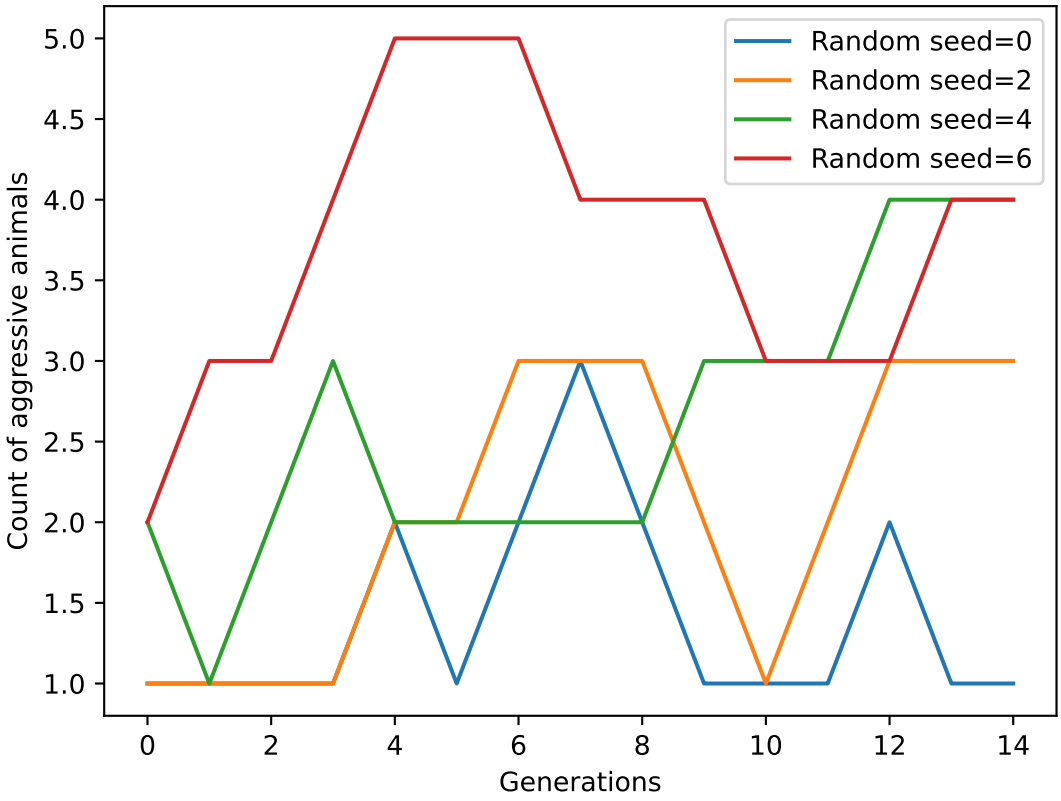
3. For $v = (6, 6)$ the probabilities are given by:

Type	Copying	Removal
Sharing	$\frac{6(5\times2+6\times1)}{6(5\times2+6\times1)+6(6\times3+5\times0)} = \frac{96}{204}$	$6/12 = 1/2$
Aggresive	$\frac{6(6\times3+5\times0)}{6(5\times2+6\times1)+6(6\times3+5\times0)} = \frac{108}{204}$	$6/12 = 1/2$

3.15.3 The Moran process with mutation

The Moran process can be modified to allow for mutation. When a new individual is selected for copying, there is a p probability that they mutate to a another type from the original population (even if they are no longer present in the population.

The following plot shows 4 possible outcomes of the Moran process of *the Hawk Dove game* with a probability of mutation of $p = .2$. Note that as opposed to the numerical simulations without mutation, the process does not terminate as new types of individuals can always enter the population.



3.15.4 The Moran process with 2 types of individuals

When considering a Moran process on 2 types of individuals the fitness function is defined by A which is, in this case is a 2 by 2 matrix.

In the case of a only two types of individuals, the population vector v can be replaced by an integer n which represents the number of individuals of the first type. The number of individuals of the second type is then given by $N - n$.

In this case the random process is a specific type of process called a birth death process:

- A set of possible states: $S = \{0, 1, \ldots, N\}$
- Two absorbing states: 0 and N.
- Probabilities p_{ij} of going from state i to j defined by:

- $p_{i,i+1} + p_{i,i-1} \leq 1$ for $1 \leq i \leq N-1$.
- $p_{ii} = 1 - p_{i,i+1} - p_{i,i-1}$ for $1 \leq i \leq N-1$.
- $p_{00} = p_{NN} = 1$.

Fixation probability

The probability of starting in state i and the process ending in state N is denoted by x_i.

The probability of a single individual of the first type being able to take over the population is denoted by ρ and $\rho = x_1$.

Given a birth death process, the probability x_i is given by:

$$x_i = \frac{1 + \sum_{j=1}^{i-1} \prod_{k=1}^{j} \gamma_k}{1 + \sum_{j=1}^{N-1} \prod_{k=1}^{j} \gamma_k}$$

where:

$$\gamma_k = \frac{p_{k,k-1}}{p_{k,k+1}}$$

The proof of this result is omitted here but it allows for the specific case of the Moran process to be obtained:

The transition probabilities are then given by:

$$p_{i,i+1} = \frac{if_1(i)}{if_1(i) + (N-i)f_2(i)} \frac{N-i}{N} \tag{3.110}$$

$$p_{i,i-1} = \frac{(N-i)f_2(i)}{if_1(i) + (N-i)f_2(i)} \frac{i}{N} \tag{3.111}$$

which gives:

$$\gamma_i = \frac{p_{i,i-1}}{p_{i,i+1}} \tag{3.112}$$

$$= \frac{\frac{(N-i)f_2(i)}{if_1(i)+(N-i)f_2(i)} \frac{i}{N}}{\frac{if_1(i)}{if_1(i)+(N-i)f_2(i)} \frac{N-i}{N}} \tag{3.113}$$

$$= \frac{(N-i)f_2(i)\frac{i}{N}}{if_1(i)\frac{N-i}{N}} \tag{3.114}$$

$$= \frac{f_2(i)}{f_1(i)} \tag{3.115}$$

Thus, the formula for x_i in the general birth death process can be used to obtain the fixation probability $\rho = x_1$.

Question

For the *hawk dove game* obtain ρ for the following population sizes:

1. $N = 2$
2. $N = 3$
3. $N = 4$

Answer

In the case of the hawk dove game we have:

$$f_1(i) = 2(i-1) + 1 \times (N-i) = N + i - 2$$
$$f_2(i) = 3i$$

1. For N = 2 we have:

	$i=1$
$f_1(i)$	1
$f_2(i)$	3
γ_i	3

   Thus:

$$\rho = \frac{1}{1+3} = \frac{1}{4}$$

2. For N = 3 we have:

	$i=1$	$i=2$
$f_1(i)$	1 + 1 = 2	1 + 2 = 3
$f_2(i)$	3	6
γ_i	3/2	6/3=2

   Thus:

$$\rho = \frac{1}{1+3/2+3/2\times 2} = \frac{1}{11/2} = \frac{2}{11}$$

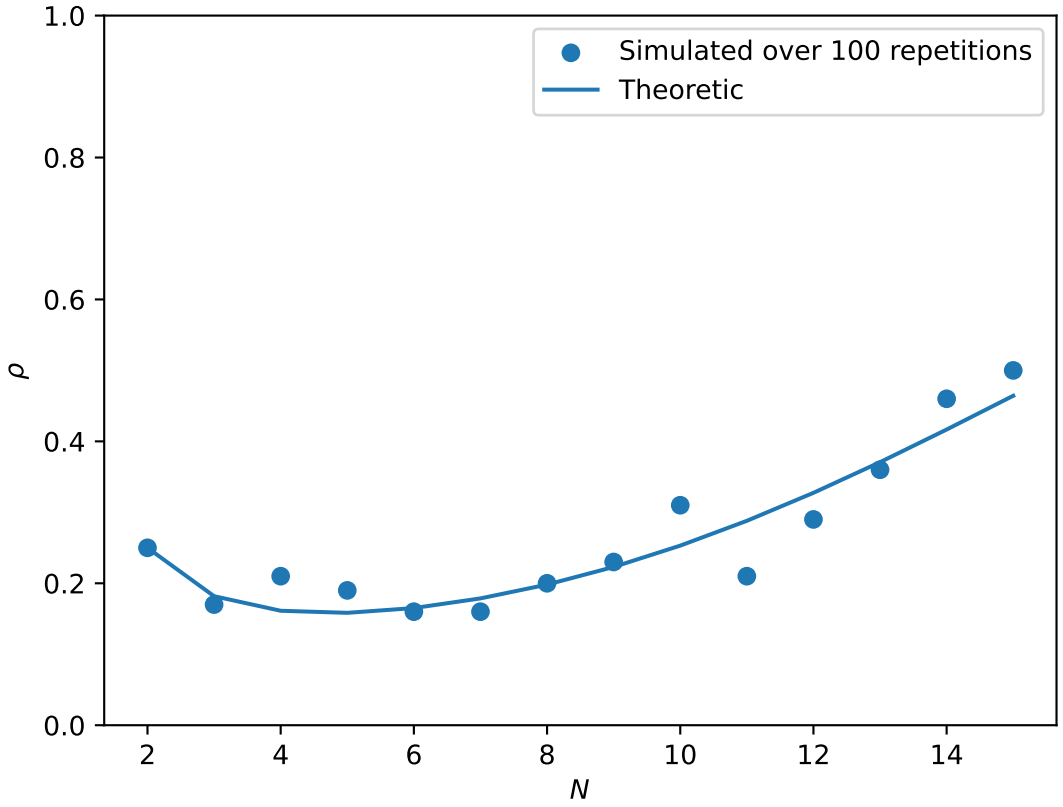
3. For N = 4 we have:

	$i=1$	$i=2$	$i=3$
$f_1(i)$	2 + 1 = 3	2 + 2 = 4	2 + 3 = 5
$f_2(i)$	3	6	9
γ_i	1	6/4=3/2	9/5

   Thus:

$$\rho = \frac{1}{1+1+1\times 3/2+1\times 3/2\times 9/5} = \frac{1}{62/10} = \frac{5}{31}$$

Below is a the fixation probability ρ for more values of N



3.15.5 Using Nashpy

See *Use Moran processes* for guidance of how to use Nashpy to obtain numerical simulations of the Moran process. See *Obtain fixation probabilities* for guidance of how to use Nashpy to obtain approximations of the fixation probabilities. This is what is used to obtain all the plots above.

CHAPTER

FOUR

REFERENCE

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 Python library that implements algorithms that are implemented 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 a Python library that only requires the common scientific python stack 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.

4.4 Bibliography

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

4.5 Source files

4.5.1 Subpackages

nash.algorithms package

Submodules

nashpy.algorithms.support_enumeration module

A class for a normal form game

nashpy.algorithms.support_enumeration.**indifference_strategies**(*A: ndarray[Any, dtype[ScalarType]]*, *B: ndarray[Any, dtype[ScalarType]]*, *non_degenerate: bool = False*, *tol: float = 1e-16*) → Generator[Tuple[bool, bool, Any, Any], Any, None]

A generator for the strategies corresponding to the potential supports

Parameters

- **A** (*array*) – The row player utility matrix.
- **B** (*array*) – The column player utility matrix
- **non_degenerate** (*bool*) – Whether or not to consider supports of equal size. By default (False) only considers supports of equal size.
- **tol** (*float*) – A tolerance parameter for equality.

Yields

Generator – 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: tuple*, *support_pair: Tuple[ndarray[Any, dtype[ScalarType]], ndarray[Any, dtype[ScalarType]]]*, *payoff_matrices: Tuple[ndarray[Any, dtype[ScalarType]], ndarray[Any, dtype[ScalarType]]]*) → bool

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

Parameters

- **strategy_pair** (*tuple*) – a 2-tuple of numpy arrays.
- **support_pair** (*tuple*) – a 2-tuple of numpy arrays of integers.
- **payoff_matrices** (*tuple*) – a 2-tuple of numpy array of payoff matrices.

Returns
True if a given strategy pair is a pair of best responses.

Return type
bool

nashpy.algorithms.support_enumeration.**obey_support**(*strategy*, *support: ndarray[Any, dtype[ScalarType]]*, *tol: float = 1e-16*) → bool

Test if a strategy obeys its support

Parameters

- **strategy** (*array*) – A given strategy vector
- **support** (*array*) – A strategy support
- **tol** (*float*) – A tolerance parameter for equality.

Returns
whether or not that strategy does indeed have the given support

Return type
bool

nashpy.algorithms.support_enumeration.**potential_support_pairs**(*A: ndarray[Any, dtype[ScalarType]]*, *B: ndarray[Any, dtype[ScalarType]]*, *non_degenerate: bool = False*) → Generator[tuple, Any, None]

A generator for the potential support pairs

Parameters

- **A** (*array*) – The row player utility matrix.
- **B** (*array*) – The column player utility matrix
- **non_degenerate** (*bool*) – Whether or not to consider supports of equal size. By default (False) only considers supports of equal size.

Yields
Generator – A pair of possible supports.

nashpy.algorithms.support_enumeration.**powerset**(*n: int*) → Iterator[Tuple[Any, ...]]

A power set of range(n)

Based on recipe from python itertools documentation:

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

Parameters
n (*int*) – The defining parameter of the powerset.

Returns
The powerset

Return type
Iterator

nashpy.algorithms.support_enumeration.**solve_indifference**(*A*, *rows=None*, *columns=None*) → Union[bool, Any]

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** (*array*) – The row player utility matrix.
- **rows** (*array*) – Array of integers corresponding to rows to consider.
- **columns** (*array*) – Array of integers corresponding to columns to consider.

Returns
The solution to the indifference equations.

Return type
Union

nashpy.algorithms.support_enumeration.**support_enumeration**(*A: ndarray[Any, dtype[ScalarType]]*, *B: ndarray[Any, dtype[ScalarType]]*, *non_degenerate: bool = False*, *tol: float = 1e-16*) → Generator[Tuple[bool, bool], Any, None]

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.

Parameters

- **A** (*array*) – The row player utility matrix.
- **B** (*array*) – The column player utility matrix
- **non_degenerate** (*bool*) – Whether or not to consider supports of equal size. By default (False) only considers supports of equal size.
- **tol** (*float*) – A tolerance parameter for equality.

Yields
Generator – The equilibria.

nashpy.algorithms.vertex_enumeration module

A class for the vertex enumeration algorithm

nashpy.algorithms.vertex_enumeration.**vertex_enumeration**(*A: ndarray[Any, dtype[ScalarType]]*, *B: ndarray[Any, dtype[ScalarType]]*) → Generator[Tuple[float, float], Any, None]

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

Parameters

- **A** (*array*) – The row player utility matrix.
- **B** (*array*) – The column player utility matrix

Yields

Generator – The equilibria.

nashpy.algorithms.lemke_howson module

A class for the Lemke Howson algorithm

nashpy.algorithms.lemke_howson.**lemke_howson**(*A: ndarray[Any, dtype[ScalarType]]*, *B: ndarray[Any, dtype[ScalarType]]*, *initial_dropped_label: int = 0*) → Tuple[ndarray[Any, dtype[ScalarType]], ndarray[Any, dtype[ScalarType]]]

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

- **A** (*array*) – The row player payoff matrix
- **B** (*array*) – The column player payoff matrix
- **initial_dropped_label** (*int*) – The initial dropped label.

Returns

An equilibria

Return type

Tuple

nashpy.algorithms.lemke_howson.**shift_tableau**(*tableau: ndarray[Any, dtype[ScalarType]]*, *shape: Tuple[int, ...]*) → ndarray[Any, dtype[ScalarType]]

Shift a tableau to ensure labels of pairs of tableaux coincide

Parameters

- **tableau** (*array*) – a tableau corresponding to a vertex of a polytope.
- **shape** (*tuple*) – the required shape of the tableau

Returns
The shifted tableau

Return type
array

nashpy.algorithms.lemke_howson.**tableau_to_strategy**(*tableau: ndarray[Any, dtype[ScalarType]]*, *basic_labels: Set[int]*, *strategy_labels: Iterable*) → ndarray[Any, dtype[ScalarType]]

Return a strategy vector from a tableau

Parameters

- **tableau** (*array*) – a tableau corresponding to a vertex of a polytope.
- **basic_labels** (*set*) – the set of basic labels.
- **strategy_labels** (*Iterable*) – the set of labels that correspond to strategies.

Returns
A strategy.

Return type
array

nash.learning package

Submodules

nashpy.learning.fictitious_play module

Code to carry out fictitious learning

nashpy.learning.fictitious_play.**fictitious_play**(*A: ndarray[Any, dtype[ScalarType]]*, *B: ndarray[Any, dtype[ScalarType]]*, *iterations: int*, *play_counts: Optional[Any] = None*) → Generator

Implement fictitious play

Parameters

- **A** (*array*) – The row player payoff matrix.
- **B** (*array*) – The column player payoff matrix.
- **iterations** (*int*) – The number of iterations of the algorithm.
- **play_counts** (*Optional*) – The play counts.

Yields
Generator – The play counts.

nashpy.learning.fictitious_play.**get_best_response_to_play_count**(*A: ndarray[Any, dtype[ScalarType]]*, *play_count: ndarray[Any, dtype[ScalarType]]*) → int

Returns the best response to a belief based on the playing distribution of the opponent

Parameters

- **A** (*array*) – The utility matrix.
- **play_count** (*array*) – The play counts.

Returns
The action that corresponds to the best response.

Return type
int

nashpy.learning.fictitious_play.**update_play_count**(*play_count: ndarray[Any, dtype[ScalarType]]*, *play: int*) → ndarray[Any, dtype[ScalarType]]

Update a belief vector with a given play

Parameters

- **play_count** (*array*) – The play counts.
- **play** (*int*) – The given play.

Returns
The updated play counts.

Return type
array

4.5.2 Submodules

4.5.3 nashpy.game module

A class for a normal form game

class nashpy.game.**Game**(**args: Any*)

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*) – The initial population distribution of the row player.

- **y0** (*array*) – The initial population distribution of the column player.
- **timepoints** (*array*) – The iterable of timepoints.

Returns
The 2 population distributions over time.

Return type
Tuple

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*) – The number of iterations of the algorithm.
- **play_counts** (*array*) – The play counts.

Returns
The play counts

Return type
Generator

fixation_probabilities(*initial_population*, *repetitions*)

Return the fixation probabilities for all types of individuals.

The returned array will have the same dimension as the number of rows or columns as the payoff matrix A. The ith element of the returned array corresponds to the probability that the ith strategy becomes fixed given the initial population.

This is a stochastic algorithm and the probabilities are estimated over a number of repetitions.

Parameters

- **initial_population** (*array*) – the initial population
- **repetitions** (*int*) – The number of iterations of the algorithm.

Returns
The fixation probability of each type.

Return type
array

is_best_response(*sigma_r*, *sigma_c*)

Checks if sigma_r is a best response to sigma_c and vice versa.

Parameters

- **sigma_r** (*array*) – The row player strategy
- **sigma_c** (*array*) – The column player strategy

Returns
A pair of booleans, the first indicates if sigma_r is a best response to sigma_c. The second indicates if sigma_c is a best response to sigma_r.

Return type
tuple

`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*) – The initial dropped label.

Returns
An equilibria

Return type
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.

Yields
Tuple – An equilibria

`moran_process`(*initial_population*, *mutation_probability=0*)

Return a generator of population across the Moran process. The last population is when only a single type of individual is present in the population.

Parameters

- **initial_population** (*array*) – the initial population
- **mutation_probability** (*float*) – the probability of an individual selected to be copied mutates to another individual from the original set of strategies (even if they are no longer present in the population).

Returns
The generations.

Return type
Generator

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

- **y0** (*array*) – The initial population distribution.
- **timepoints** (*array*) – The iterable of timepoints.

- **mutation_matrix** (*array*) – The mutation rate matrix. Element [i, j] gives the probability of an individual of type i mutating to an individual of type j. Default behaviour is to be the identity matrix which corresponds to no mutation.

Returns
The population distributions over time.

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*) – The number of iterations of the algorithm.
- **play_counts** (*array*) – The play counts.
- **etha** (*float*) – The noise parameter for the logit choice function.
- **epsilon_bar** (*float*) – The maximum stochastic perturbation.

Returns
The play counts

Return type
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.

Parameters

- **non_degenerate** (*bool*) – Whether or not to consider supports of equal size. By default (False) only considers supports of equal size.
- **tol** (*float*) – A tolerance parameter for equality.

Returns
The equilibria.

Return type
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
The equilibria.

Return type
generator

4.5.4 Module contents

A library with algorithms on 2 player games.

CHAPTER

FIVE

CONTRIBUTOR DOCUMENTATION

This is guide explains the various tools and steps used to contribute code to Nashpy.

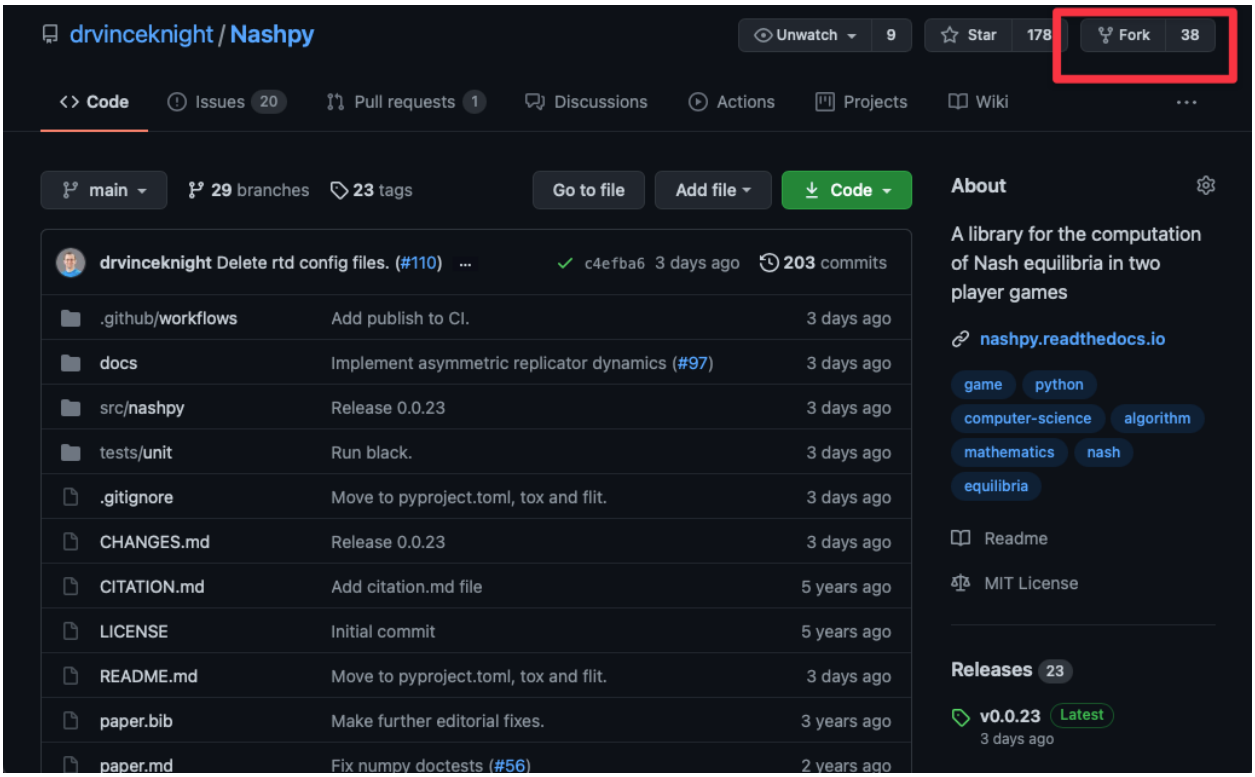
5.1 Tutorial: make a contribution to the documentation

In this tutorial we will make a contribution to the documentation of Nashpy.

5.1.1 Forking the repository

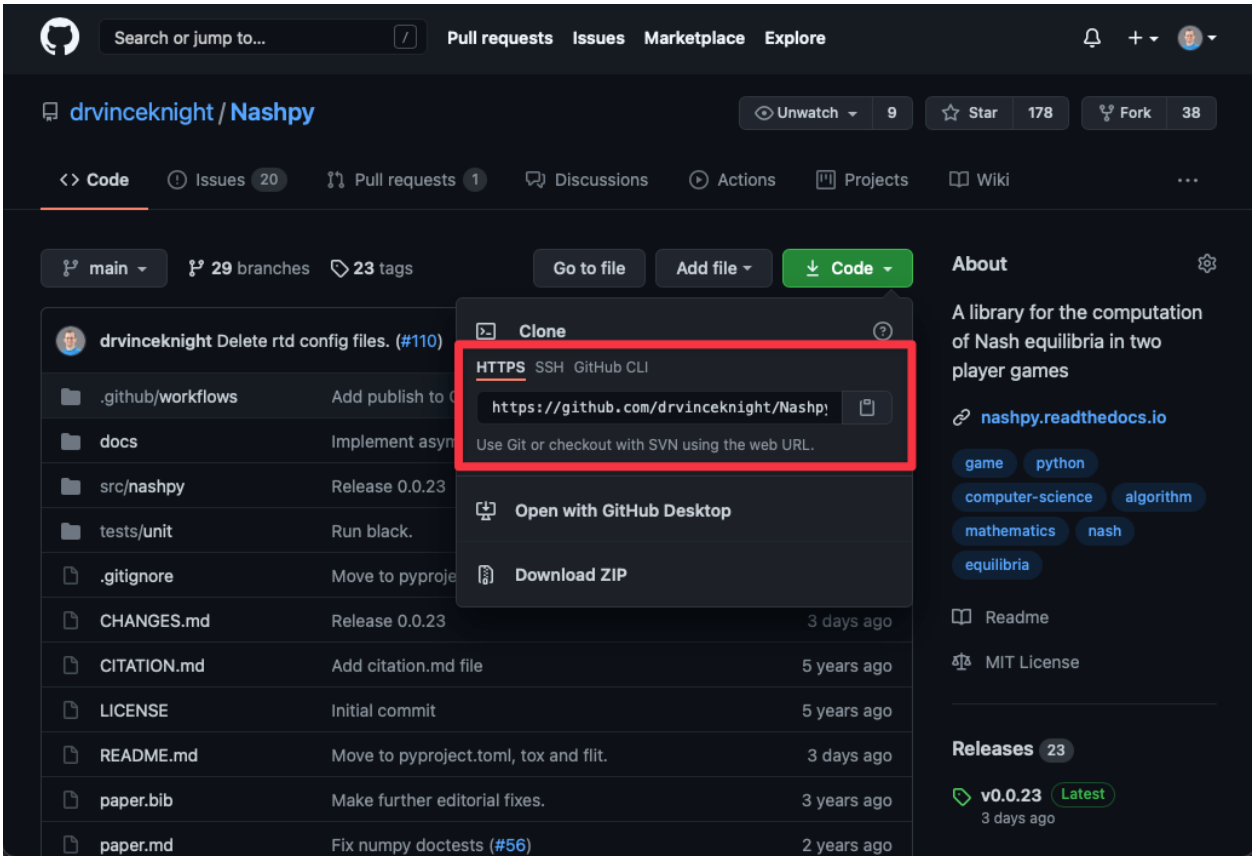
Navigate to http://github.com and create an account. If you are in education you can apply for a specific education account here: https://education.github.com.

Navigate to the Github repository for Nashpy: https://github.com/drvinceknight/Nashpy. This is the hub for development of the source code. You cannot make modification to this copy of the source code so you need to create your own copy under your Github account. You do this by creating a **fork**. Do this by clicking the `Fork` button and following the instructions:



5.1.2 Cloning the repository

Once we have a fork of the repository on **your** Github account, create a copy of it to your computer. This is called cloning. Do this by clicking the *Code* button and copying the address of the repository to your clipboard:



If you have not installed `git` go to https://git-scm.com and install.

Now to create a clone of the source code open your command line tool and type the following (**replace** <your username> with your Github username):

> $ git clone https://github.com/<your username>/Nashpy.git

This will download the source code to your computer:

```
$ git clone https://github.com/<your username>/Nashpy.git
Cloning into 'Nashpy'...
remote: Enumerating objects: 1813, done.
remote: Counting objects: 100% (362/362), done.
remote: Compressing objects: 100% (225/225), done.
remote: Total 1813 (delta 160), reused 233 (delta 79), pack-reused 1451
Receiving objects: 100% (1813/1813), 439.94 KiB | 2.67 MiB/s, done.
Resolving deltas: 100% (905/905), done.
```

5.1.3 Creating a branch

In order to modify the source code you must create a new branch. After cloning, first change directory in to the Nashpy source code:

```
$ cd Nashpy
```

Now, to keep the changes you are about to make separate from the `main` source code, create a **branch**:

```
$ git branch add-name-to-contributors-list
```

Now checkout to that branch:

```
$ git checkout add-name-to-contributors-list
```

5.1.4 Modifying the documentation

Using your preferred editor, open the file `Nashpy/docs/contributing/reference/contributors/index.rst`. If you do not have a preferred editor Visual Studio Code is recommended.

Now add you name to the file (**replace** <your username> with your Github username):

```
List of contributors
--------------------

- `@drvinceknight <https://github.com/drvinceknight>`_
- `@<your username> <https://github.com/<your username>`_
```

5.1.5 Checking the modification

To build the documentation, first create a virtual environment specifically for purposes to work on Nashpy:

```
$ python -m venv env
```

This creates a directory `env` which holds a separate version of python. To tell your command line tool to now use this version:

On Linux or macOS type:

```
$ source env/bin/activate
```

On Windows type:

```
$ env\Scripts\activate
```

Now install the Nashpy software in to this environment:

```
$ python -m pip install flit
$ python -m flit install --symlink
```

To build the documentation:

```
$ cd docs
$ sphinx-build -b html . _build/html
Running Sphinx v3.1.2
loading pickled environment... done
building [mo]: targets for 0 po files that are out of date
building [html]: targets for 2 source files that are out of date
updating environment: 1 added, 2 changed, 1 removed
reading sources... [100%] contributing/tutorial/index
looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [100%] index
generating indices...  genindex py-modindexdone
highlighting module code... [100%] nashpy.learning.fictitious_play
writing additional pages...  searchdone
copying images... [100%] _static/contributing/tutorial/cloning/main.png
copying static files... ... done
copying extra files... done
dumping search index in English (code: en)... done
dumping object inventory... done
build succeeded.

The HTML pages are in _build/html.
```

You can open `_build/html/index.html` in a browser to see the documentation locally which should include the changes you made.

5.1.6 Running the test suite

You can run the entire test suite which will check that this modification has not caused any problems:

```
$ python -m pip install tox
$ python -m tox
```

5.1.7 Committing the change

Now you need to **stage** this file:

```
$ git add docs/contributing/reference/contributors/index.rst
```

Now commit this file:

```
$ git commit
```

This will open a text editor where you can write your commit title and message:

```
Add <your username> to list of contributors

I am doing the contribution tutorial.
```

Closing the editor will commit the changes you made.

5.1.8 Pushing the change to Github

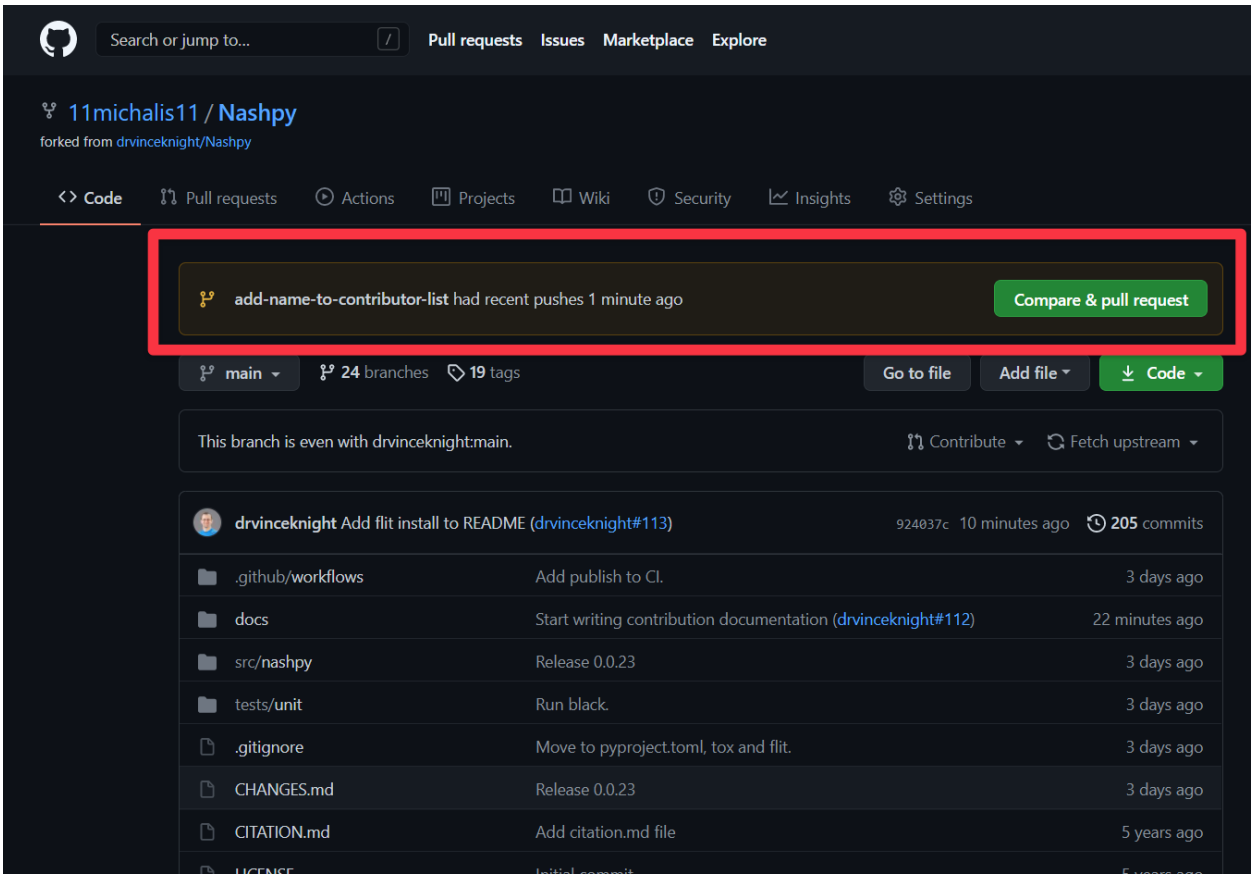
Now that all that is done, you are going to send the changes back to your copy of the source code on Github:

```
$ git push origin add-name-to-contributors-list
```

5.1.9 Opening a Pull Request

You now have 2 copies of the modified source code of Nashpy. One locally on your computer, the other under your Github account. In order to include those changes in to the main source code of Nashpy you will open a Pull request.

To do this, go to your fork of the Nashpy repository: `https://github.com/<your username>/Nashpy`. You should see a `Compare and Pull Request` button:

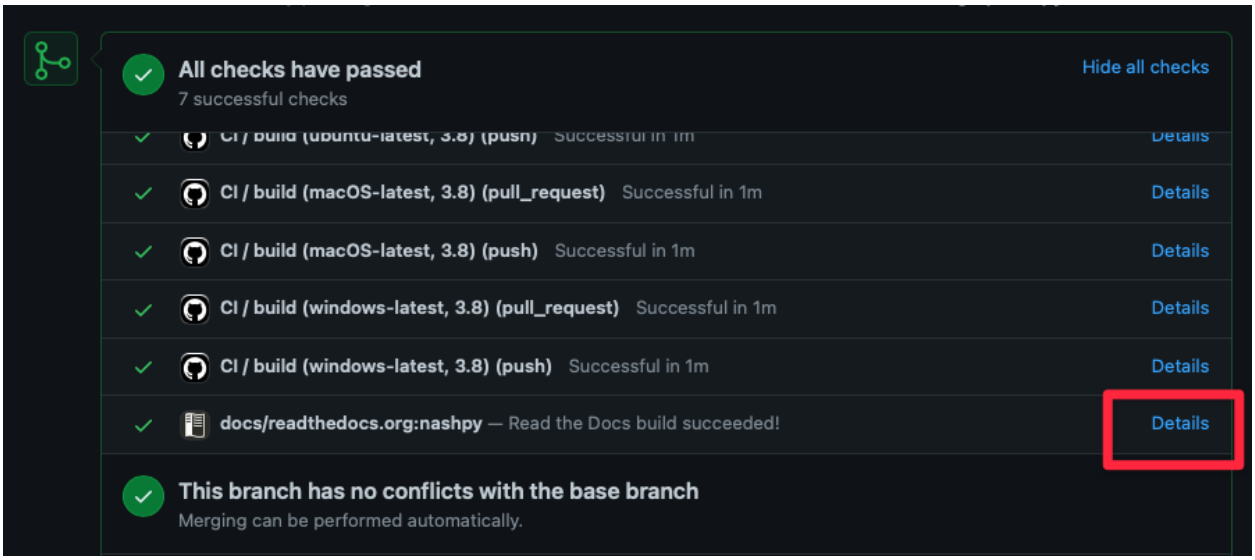


Once you have clicked on that, you can review your changes and then eventually click on `Create pull request` to create the Pull Request.

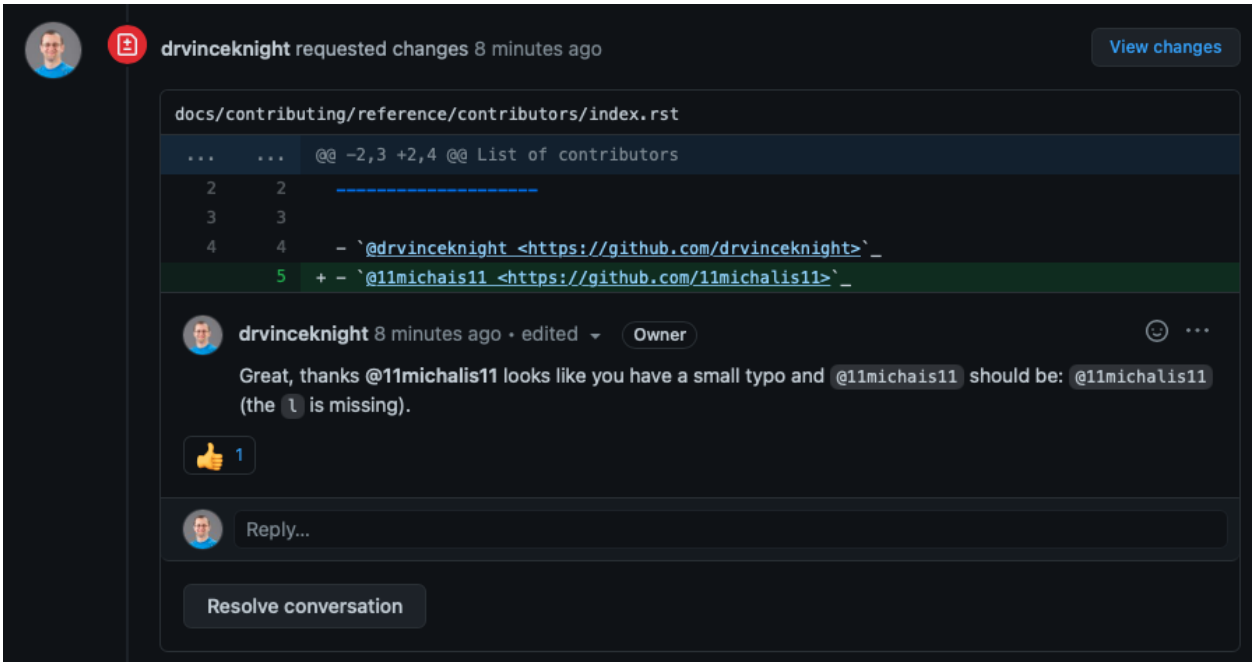
5.1.10 Making further modifications

Once a Pull Request is opened, a number of automated checks will start. This will check the various software tests but also build a viewable version of the documentation.

You can click on the corresponding `details` button to see any of these:



Your modification will also be reviewed:



To make any required changes, **modify the files**.

Then stage and commit the files:

```
$ git add docs/contributing/reference/contributors/index.rst
$ git commit
```

This will open a text editor where you can write your commit title and message (similarly to before).

Once this is done, push the code to Github which will automatically update the pull request:

```
$ git push origin add-name-to-contributors-list
```

This final process of making further modifications might repeat itself and eventually the Pull Request will be **merged** and your changes included in the main version of the Nashpy source code.

5.2 How to

How to:

5.2.1 How to fork the repository

1. Navigate to https://github.com/drvinceknight/Nashpy
2. Click on *Fork*
3. Follow any remaining instructions, for example if you have an organisation account on Github you will be prompted to say which account you would like to fork with.

5.2.2 How to clone the repository

In order to obtain a copy of *your fork of the repository on github* to your local machine run the following at your command line:

```
$ git clone https://github.com/<username>/Nashpy.git
```

5.2.3 How to update from upstream

In order to bring your local repository up to date with any upstream changes:

```
$ git remote add upstream https://github.com/drvinceknight/Nashpy.git
$ git pull upstream main
```

The above:

1. Creates a new remote with alias `upstream` that points at the main source repository for Nashpy. You can list all your remote repositories by running:

   ```
   $ git remote -v
   ```

2. Pulls the latest changes from the `main` branch of the upstream repository.

5.2.4 How to create a branch

To create a branch with name <name-of-branch> run:

```
$ git branch <name-of-branch>
```

To go to that branch:

```
$ git checkout <name-of-branch>
```

How to branch from a specific branch, tag or commit

To create a branch with name <name-of-branch> from a specific branch, tag or commit with name <location> run:

```
$ git branch <name-of-branch> <location>
```

How to create a branch and checkout to it at the same time

You can create a branch with name <name-of-branch> and checkout in a single command:

```
$ git checkout -b <name-of-branch>
```

5.2.5 How to create a virtual environment

To create a virtual environment with name <name> run:

```
$ python -m venv <name>
```

How to activate <name>

On Linux or macOS type:

```
$ source env/bin/activate
```

On Windows type:

```
$ env\Scripts\activate
```

5.2.6 How to install the library from source

To install the library from the source files:

```
$ python -m pip install flit
$ python -m flit install
```

How to install an editable version of the library from source

If you want to install a version of the library from the source files so that modifications of the source files are directly usable:

```
$ python -m flit install --symlink
```

5.2.7 How to run tests

To install `tox`:

```
$ python -m pip install tox
```

To run all tests:

```
$ python -m tox
```

If you want to run the tests across a single version of Python:

```
$ python -m tox -e <version>
```

where `version` is either `py38` or `py39`.

5.2.8 How to check for insensitive language

The documentation is checked for insensitive or inconsiderate language using alex.

How to install alex

To install alex run:

```
$ npm install alex --global
```

Note that this required node, information on install node is available here: https://www.npmjs.com/get-npm

How to run alex

To run alex on the documentation:

```
$ alex docs**/*.rst
```

To run alex on the `README.md` file:

```
$ alex README.md
```

How to ignore some checks

To ignore some specific checks annotations can be used. For example *John Nash* is annotated to ignore insensitive related to gendered pronouns:

```
.. <!--alex disable he-she-->
.. <!--alex disable her-him-->

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.

..  <!--alex enable he-she-->
..  <!--alex enable her-him-->
```

Another example is the `README.md` file where an annotation is added to ignore a specific use of the word "bi":

```
## Usage

Create bi-matrix games by passing two 2 dimensional arrays/lists:
```

5.2.9 How to write a docstring

All functionality needs to have a documentation string (docstrings). The convention used in Nashpy is to follow Numpy's docstring convention:

```
def <function>(<signature>):
    """
    <short summary>

    Parameters
    ----------
    <paramter> : <type>
        <description>
    <paramter> : <type>
        <description>
    ...
    <paramter> : <type>
        <description>

    Returns
    -------
    <type>
        <description>
    """
```

If the function/method does not return anything but is instead a **generator** then `Returns` should be replaced with `Yields`.

How to check docstrings in a module

Running tests with *tox* will automatically check formatting of docstrings.

If you want to check a specific file, use darglint:

```
$ python -m pip install darglint
$ darglint -s numpy <path_to_file>
```

5.2.10 How to write a type hint

Type hints allow to annotate code in a machine and human readable way so as to indicate the types of each variable. The general syntax of this is:

```
def <function>(
    variable_1: type = default_value,
    variable_2: type,
    ) -> type
```

For example here is the annotated source code for some internal functionality:

```
import numpy as np
import numpy.typing as npt


def make_tableau(M: npt.NDArray) -> npt.NDArray:
    """
    Make a tableau for the given matrix M.
    This tableau corresponds to the polytope of the form:
       Mx <= 1 and x >= 0
    Parameters
    ----------
    M : array
        A matrix with linear coefficients defining the polytope.
    Returns
    -------
    array
        The tableau that corresponds to the polytope.
    """
    return np.append(
        np.append(M, np.eye(M.shape[0]), axis=1),
        np.ones((M.shape[0], 1)),
        axis=1,
    )
```

How to check type annotations in a module

Running tests with *tox* will automatically check type annotations.

If you want to check a specific file, use Mypy:

```
$ python -m pip install mypy
$ python -m mypy --ignore-missing-imports <path_to_file>
```

5.2.11 How to write tests

The *pytest* framework is used for writing and running tests for Nashpy.

Tests should be written in one of the following locations:

- In a preexisting file in the `test/` directory.
- In a new file in the `test/` directory.

Thanks to `pytest` the format for a test is:

```
def test_<functionality>():
    """
    <short summary if necessary>
    """
    <code logic>
    assert <boolean>
```

For guidance on how to run tests see: *How to run tests*.

When writing a new test it is good practice to ensure the test fails (either by modifying the test or by modifying the source code): this ensures that *pytest* is running the test in question.

Note that when adding new functionality the coverage of the test suite will be checked using *coverage*. Thus, in practice multiple tests will need to be written to test new functionality completely.

Hypothesis

Property based tests are tests that use random sampling in an efficient manner to test given properties as opposed to specific values. Nashpy uses *hypothesis* for this.

For example the following tests that for any given `M`, which is a 3 by 3 numpy integer array, the length of the output of `get_derivative_of_fitness` is as expected:

```
from hypothesis import given, settings
from hypothesis.strategies import integers
from hypothesis.extra.numpy import arrays

@given(M=arrays(np.int8, (3, 3)))
def test_property_get_derivative_of_fitness(M):
    t = 0
    x = np.zeros(M.shape[1])
    derivative_of_fitness = get_derivative_of_fitness(x, t, M)

    assert len(derivative_of_fitness) == len(x)
```

5.2.12 How to make a commit

To commit changes to a given code <file.py>.

First, stage it:

```
$ git add <file.py>
```

Now that the file is staged, create a commit:

```
$ git commit
```

This will open a text editor (a default text editor of your choice can be set). In there write a commit message in the following format:

```
<commit title>

<commit message>
```

Save and exit from the text editor and your commit should be applied.

Commit message style

The <commit title> should be short and follow the style: "If this commit is applied it will <commit title> will happen."

The <commit message> should include further details and can go over many lines.

Here is some good guidance on writing commit messages: https://chris.beams.io/posts/git-commit/

Do not use `git commit -m "<commit message>"`

It is possible to write a commit message directly as you make the commit by typing:

```
$ git commit -m <commit title>
```

This is not recommended as it encourages unclear commit messages.

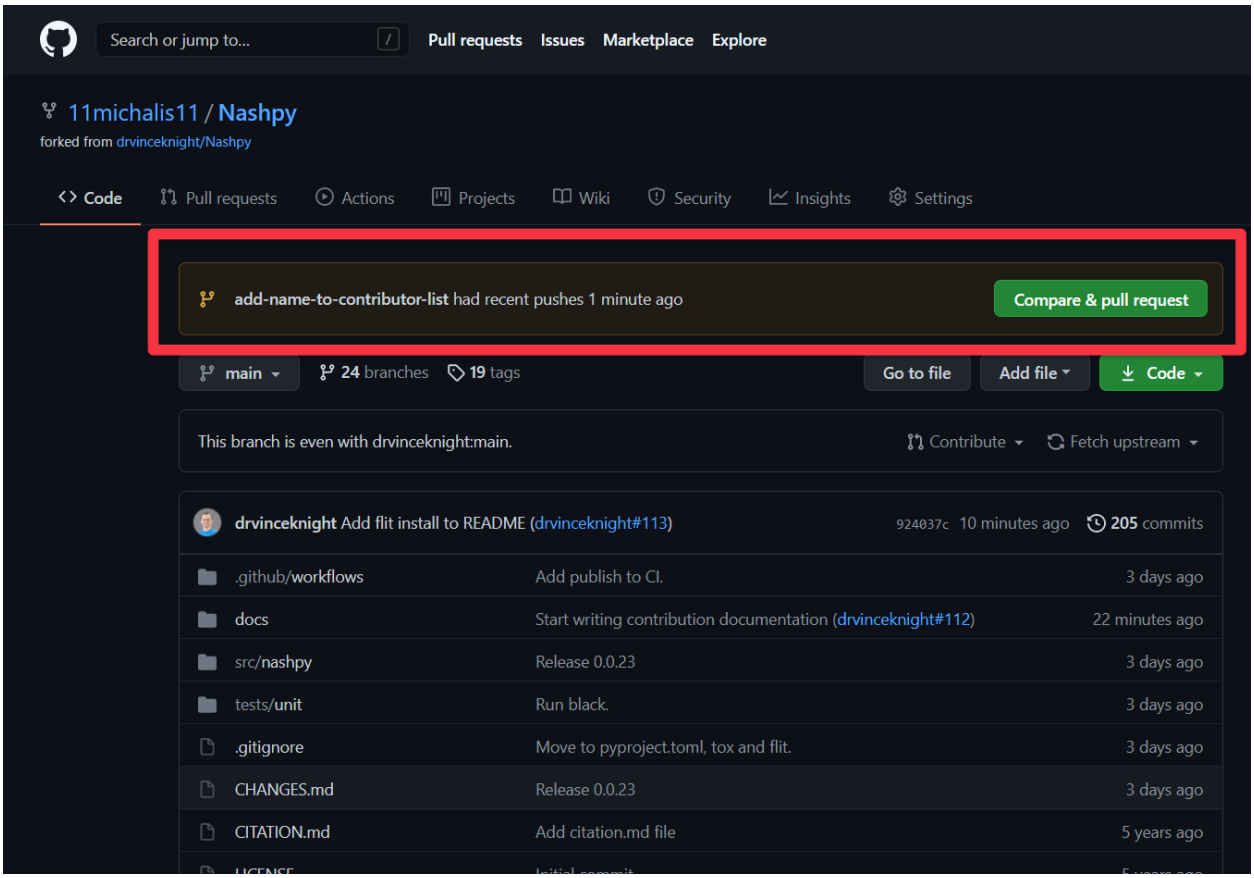
5.2.13 How to push changes

In order to push a copy of your committed changes on the <branch-name> branch to *your fork of the repository on github* run the following at your command line:

```
$ git push origin <branch-name>
```

5.2.14 How to open a pull request

Once you have *pushed your changes to github* you can open a request for these changes to be incorporated in to the main repository by going to your fork of the Nashpy repository: `https://github.com/<your username>/Nashpy`. You should see a `Compare and Pull Request` button:



Once you have clicked on that, you can review your changes and then eventually click on `Create pull request` to create the Pull Request.

5.2.15 How to format markdown files

The markdown files are all formatted according to a consistent style using mdformat.

How to install mdformat

To install `mdformat` run:

```
$ python -m pip install mdformat
```

How to format all markdown files

To run `mdformat` on a specific `<file>`:

```
$ python -m mdformat <file>
```

This will reformat `<file>` in place.

To run `mdformat` recursively from the current directory:

```
$ python -m mdformat .
```

How to check markdown files

To check the format of `<file>`:

```
$ python -m mdformat --check <file>
```

5.3 Discussion

5.3.1 The code structure of Nashpy

The directory structure

The directory structure for Nashpy is:

```
├── src/
├── tests/
├── docs/
├── CHANGES.md
├── CITATION.md
├── LICENSE
├── README.md
├── paper.bib
├── paper.md
├── pyproject.toml
├── .readthedocs.yml
├── setup.cfg
└── tox.ini
```

Here is a brief description of each of these:

The src/ directory

The src/ directory contains the source code. It's structure is as follows:

```
src/
└── nashpy
    ├── __init__.py
    ├── game.py
    ├── algorithms/
    ├── integer_pivoting/
    ├── learning/
    └── polytope/
```

- The __init__.py file contains the various commands to import all the functionality of the library.
- The game.py file contains the main nashpy.Game class.
- The algorithms/ directory contains further modules with algorithms for computation of Nash equilibria.
- The integer_pivoting/ directory contains further modules with algorithms for integer pivoting.
- The learning/ directory contains further modules for various learning algorithms.
- The polytope directory contains further modules with code for *best response polytopes*.

The tests/ directory

This contains all the test files.

The docs/ directory

The documentation is written using the Diataxis framework [Procida2021]. As well as various configuration files for *sphinx* there are 5 main subdirectories:

```
docs/
├── contributing
│   ├── discussion/
│   ├── how-to/
│   ├── index.rst
│   ├── reference/
│   └── tutorial/
├── discussion/
├── how-to/
├── index.rst
├── reference/
└── tutorial/
```

- The contributing/ directory contains the specific contributing documentation. Which itself is written using Diataxis [Procida2021].
- The discussion/ directory contains source files for the discussion described at [Procida2021] as: "explanation is discussion that clarifies and illuminates a particular topic."
- The reference/ directory contains source files for the reference described at [Procida2021] as: "reference guides are technical descriptions of the machinery and how to operate it."

- The `how-to/` directory contains source files for the how to guides described at [Procida2021] as: "how-to guides are directions that take the reader through the steps required to solve a real-world problem"
- The `tutorial/` directory contains source files for the tutorial described at [Procida2021] as: "tutorials are lessons that take the reader by the hand through a series of steps to complete a project of some kind."

The `CHANGES.md` file

Makes a note of different changes in versions of Nashpy.

The `CITATION.md` file

Contains information for citing Nashpy.

The `LICENSE` file

Contains the license.

The `README.md` file

Contains the first entry point documentation to the Nashpy project.

The `paper.bib` and `paper.md` files

These are the source files for the Journal of Open Source Software paper written about Nashpy: [Knight2018].

The `pyproject.toml` file

Contains all the build instructions for packaging Nashpy and is used by *flit*.

The `.readthedocs.yml` file

This includes configuration settings for the online service that hosts the documentation *read the docs*.

The `setup.cfg` file

Contains some configuration instructions for testing.

The `tox.ini` file

Contains the instructions for the test runner `tox`.

The Game class

The `nashpy.Game` class is an umbrella class that creates an object oriented interface to all functionality of Nashpy as methods on a game.

5.3.2 Writing clean tests with pytest

The pytest framework allows for cleaner tests to be written but also for efficient running of tests with multiple plugins.

Plugins

Coverage: `pytest-cov`

The pytest-cov plugin allows you to run coverage checks with pytest.

Flake8: `pytest-flake8`

The pytest-flake8 plugin allows you to run flake8 checks with pytest.

Stochastic effects: `pytest-randomly`

The pytest-randomly plugin does two things (for Nashpy):

1. It randomly shuffles the order of tests: this ensures that tests passing is not dependent on the order in which they run.
2. It seeds stochastic tests to ensure that any exceptions are reproducible. In practice this has little effect here as ideally stochastic tests are seeded or written with *hypothesis*.

Nicer look: `pytest-sugar`

The pytest-sugar plugin changes the look of pytest.

Further plugins

The *Talk Python to Me* podcast episode 267 featured a discussion of a number of pytest plugins: https://talkpython.fm/episodes/show/267/15-amazing-pytest-plugins

5.3.3 Testing across environments with tox

The tox project allows for the automation of many tasks related to Python packaging and testing.

For Nashpy it is used to:

1. Configure all tests.
2. Test across multiple python versions.

Configure all tests

All test commands are written in `tox.ini`. This include things like checking style with *black* and presence of docstrings with *interrogate*. Running all the checks is done with a single standard command: `python -m tox`.

Note that *checking for insensitive language in documentation* is not configured or run by tox.

Test across multiple python versions

This is done thanks to configurations written in `tox.ini`:

```
[tox]
isolated_build = True
envlist = py38, py39
```

5.3.4 Installing and packaging with flit

TODO

5.3.5 Virtual environments

TODO

5.3.6 Checking code is tested with coverage

TODO

5.3.7 Testing with properties with hypothesis

TODO

5.3.8 Ensuring consistent code style with Black

TODO

5.3.9 Static code analysis with flake8

TODO

5.3.10 Checking the presence of docstrings with interrogate

TODO

5.3.11 Checking the format of docstrings with darglint

Documentation strings, more commonly referred to as docstrings in python are strings that directly document a function. Their presence is checked using *Checking the presence of docstrings with interrogate* but the particular format they are written in is checked using darglint.

Once installed darlint can be used to check one of three docstring styles:

1. Google style guide
2. Sphinx style guide
3. Numpy style guide

For example, consider the file `main.py`:

```
def get_mean(collection):
    """
    Obtain the average of a collection of objects.

    Parameters
    ----------
    collection : list
        A list of numbers

    Returns
    -------
    float
        The mean of the numbers.
    """
    return sum(collection) / len(collection)
```

After installing darglint:

```
$ python -m pip install darglint
```

If we check the format of this file against the Google style guide:

```
$ darglint -s google main.py
main.py:get_mean:1: DAR101: - collection
main.py:get_mean:1: DAR201: - return
```

we get two errors, we can cross reference the error codes `DAR101` and `DAR201` at https://github.com/terrencepreilly/darglint#error-codes.

- `DAR101`: "The docstring is missing a parameter in the definition."
- `DAR201`: "The docstring is missing a return from definition."

Note that our file does have both those things but here darglint is telling us that they do not match with the google style guide.

If we check the format of this file against the Sphinx style guide:

```
$ darglint -s sphinx main.py
main.py:get_mean:1: DAR101: - collection
main.py:get_mean:1: DAR201: - return
```

we get the same two errors.

Running, the file against the Numpy style guide gives:

```
$ darglint -s numpy main.py
$
```

No errors are raised as this is indeed written using the Numpy style guide which is also the convention chosen for the entire Nashpy source code.

Running darglint as part of the test suite

If darglint is installed it will automatically run as part of the flake8 check. For Nashpy this is done as part of the pytest run which is all configured using tox.

5.3.12 Checking of type hints using mypy

Optional type hints can be added to python code which allows specification of the type of a variable. Type hints were specified in PEP484.

Type hints are ignored when running the code but can be statically analysed using a various tools:

- Mypy.
- Pyright

Mypy is used for Nashpy.

For example, consider the file `main.py`:

```
def get_mean(collection):
    """
    Obtain the average of a collection of objects.

    Parameters
    ----------
    collection : list
        A list of numbers

    Returns
    -------
```

(continues on next page)

(continued from previous page)

```
    float
        The mean of the numbers.
    """
    return sum(collection) / len(collection)
```

After installing mypy:

```
$ python -m pip install mypy
```

If we check the annotations present in the file:

```
$ python -m mypy main.py
Success: no issues found in 1 source file
```

There are no issues because there are no annotations. If the following annotations are added:

```
from typing import Iterable


def get_mean(collection: Iterable) -> float:
    """
    Obtain the average of a collection of objects.

    Parameters
    ----------
    collection : Iterable
        A list of numbers

    Returns
    -------
    float
        The mean of the numbers.
    """
    return sum(collection) / len(collection)
```

We get:

```
$ python -m mypy main.py
main_with_wrong_types.py:17: error: Argument 1 to "len" has incompatible type
↪"Iterable[Any]"; expected "Sized"
Found 1 error in 1 file (checked 1 source file)
```

Mypy has found an error here: the `Iterable` type does not necessarily have a length. The following modifies this:

```
def get_mean(collection: list) -> float:
    """
    Obtain the average of a collection of objects.

    Parameters
    ----------
    collection : list
        A list of numbers

```

(continues on next page)

(continued from previous page)

```
    Returns
    -------
    float
        The mean of the numbers.
    """
    return sum(collection) / len(collection)
```

We get:

```
$ python -m mypy main.py
Success: no issues found in 1 source file
```

–ignore-missing-import

In some cases some imported modules cannot be used checked with Mypy, these can be ignored by running the following:

```
$ python -m mypy --ignore-missing-import main.py
```

Overlap of functionality with darglint

The python library darglint checks the format of the docstrings. This will also use any type annotations and so the type annotations and the types specified in the docstrings must correspond.

5.3.13 Using sphinx for documentation

TODO

Using sphinx-togglebutton for the questions

TODO

Using matplotlib for plotting directives

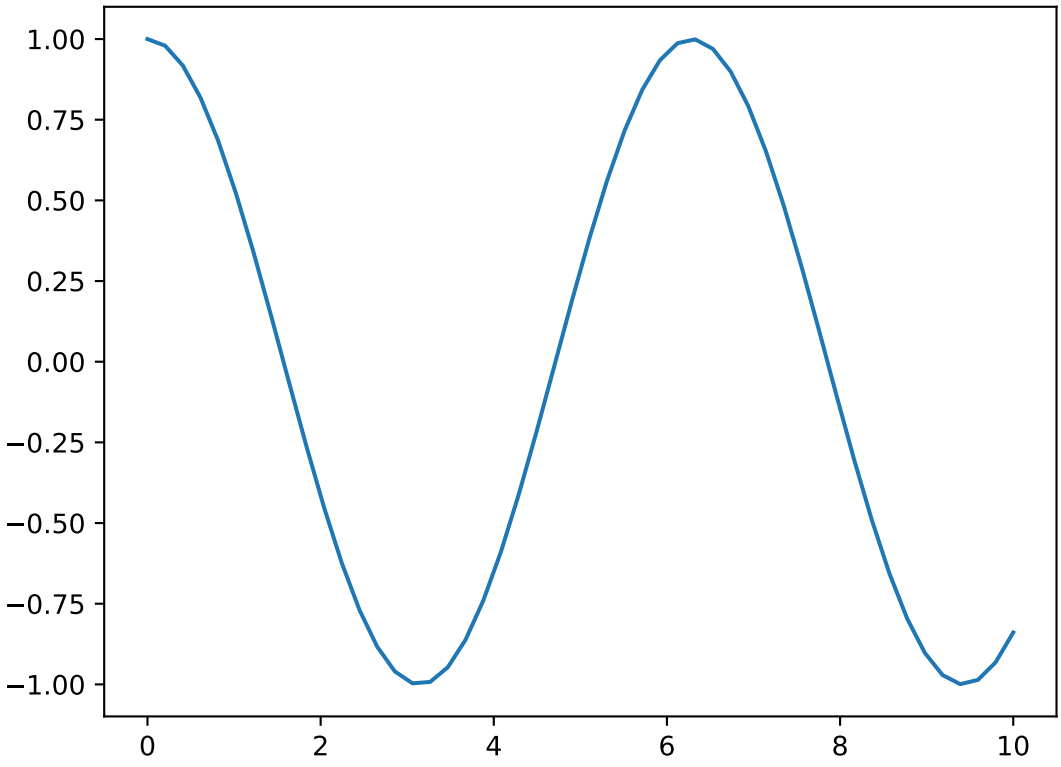
The `matplotlib` library includes a `sphinx` plugin that allows for plot directives. To enable it, ensure that "`matplotlib.sphinxext.plot_directive`" is included in `extensions` in `conf.py`.

For example the following will create a plot:

```
.. plot::

   import matplotlib.pyplot as plt
   import numpy as np

   xs = np.linspace(0, 10)
   plt.plot(xs, np.cos(xs))
```


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10

Using mermaid for diagrams

A popular tool for drawing diagrams is mermaid.js. This can be used directly with `sphinx`. To enable it, ensure that `"sphinxcontrib.mermaid"` is included in `extensions` in `conf.py`.

For example the following will create a flowchart:

```
.. mermaid::

   graph TD;
      A-->B;
      A-->C;
      B-->D;
      C-->D;
```

5.3.14 Ensuring the code in the documentation is correct with doctests

TODO

5.3.15 Checking for insensitive language with alex

alex is a tool that allows you to identify insensitive and/or inconsiderate language in written prose. The following description is taken from the project page:

> "Whether your own or someone else's writing, alex helps you find gender favoring, polarizing, race related, religion inconsiderate, or other unequal phrasing in text."

As an example consider the following markdown file:

```
# A typical user of Nashpy

He will use it to study games.
```

If we run alex on it:

```
$ alex main.md
```

We get:

```
$ alex main.md
main.md
3:1-3:3  warning  `He` may be insensitive, use `They`, `It` instead  he-she  retext-
↪equality

 1 warning
```

Correcting the markdown file to:

```
# A typical user of Nashpy

They will use it to study games.
```

Running alex now gives:

```
$ alex main.md
main.md: no issues found
```

The above example is quite a clear one, alex assists by identifying errors like this but also more subtle ones.

It is possible to ignore certain checks using a configuration file but as described in *the how to guide* it is also possible to annotate the file itself. This is preferred as it makes exceptions explicit.

FAQ

The Frequently asked questions about alex can be found here: https://github.com/get-alex/alex#faq This includes:

> Q: This is stupid!
>
> A: Not a question. And yeah, alex isn't very smart. People are much better at this. But people make mistakes, and alex is there to help

The Nashpy library uses alex for exactly this reason, it is one of many efforts made to ensure the project is inclusive.

5.3.16 Using Github Actions to check automatically run all checks and publish new releases

TODO

5.3.17 Hosting documentation on Read The Docs

Read the docs is a web service that builds and hosts documentation. You can read more about the service here: https://readthedocs.org

The documentation contained in `docs/` is automatically built and can be viewed at https://nashpy.readthedocs.io/en/stable/.

Versions

The default version (ie when going to https://nashpy.readthedocs.io/) is the `stable` version which means the last release.

You can view the version of the documentation currently on the `main` branch by going to: https://nashpy.readthedocs.io/latest.

Configuration file

Read the docs can have specific settings set in a `.readthedocs.yml` file. Details on this can be found here: https://docs.readthedocs.io/en/stable/config-file/v2.html#packages

One specific setting used by Nashpy is:

```
python:
   version: 3.8
   install:
     - method: pip
       path: .
       extra_requirements:
         - doc
```

This ensures Read the docs does not look for a `requirements.txt` file to install the library. Instead it runs `pip install .` which uses *The pyproject.toml file*. It also uses the extra requirements specified by `[doc]` in the `pyproject.toml` file.

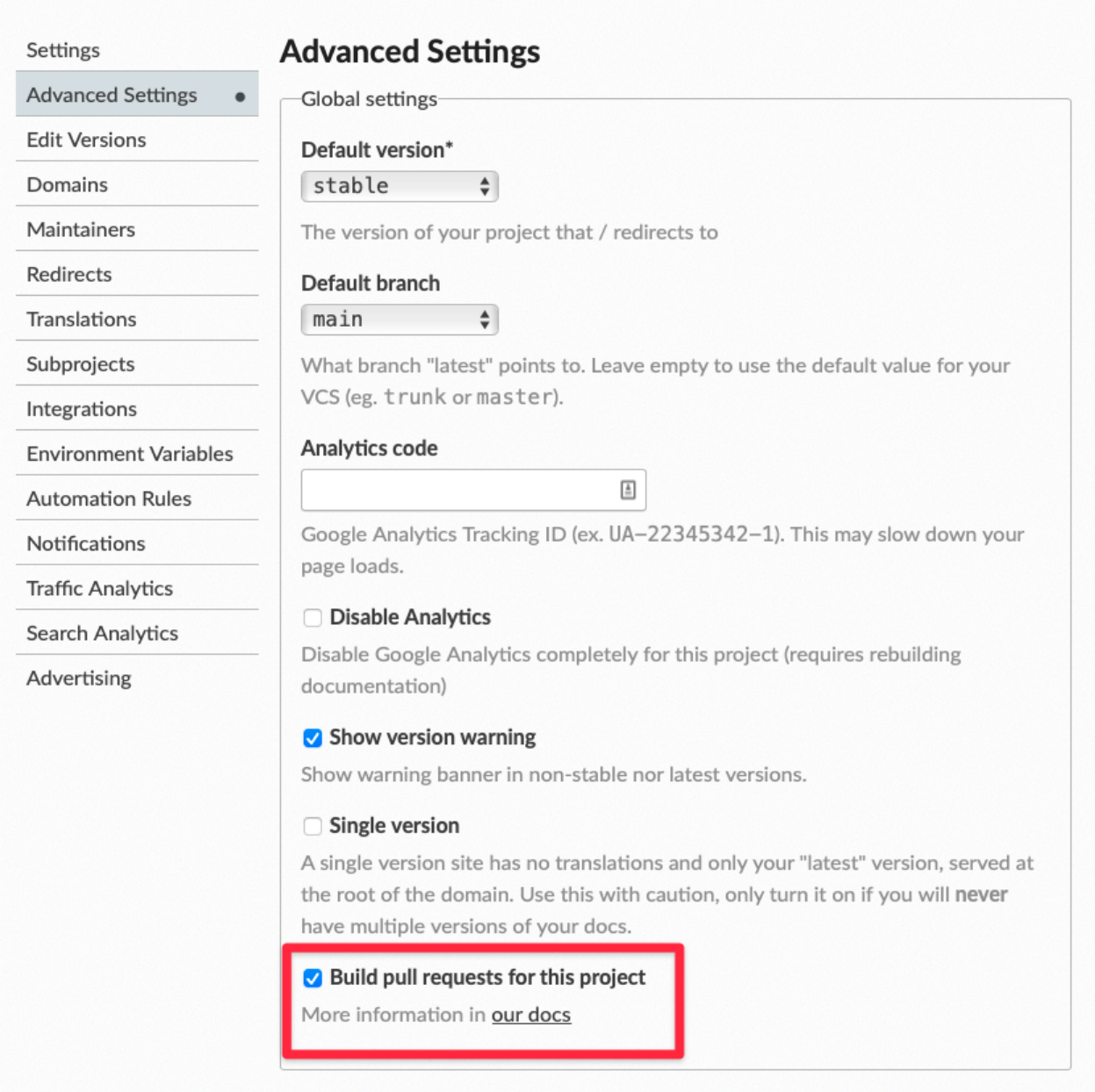
A powerful feature offered by Read the docs is that it can build documentation in pull requests.

Building documentation in pull requests

To set this up you need to ensure the following things are done:

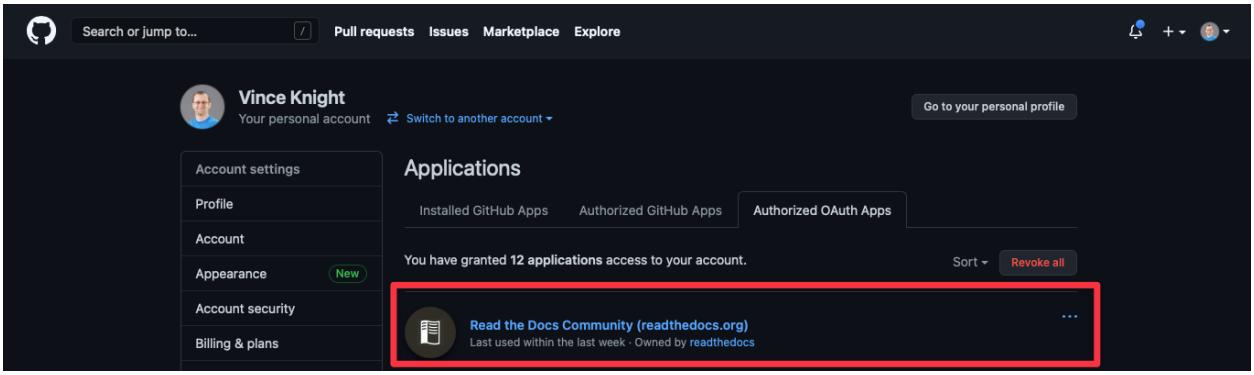
1. The repository settings on Read the docs instruct pull requests to be built.
2. The correct web hook is in place on Github.
3. The correct settings of the web hook are done on the Github repository.

To instruct pull requests to be built ensure the following box is ticked in the Advanced settings for your project on Read the docs:



Setting up the web hooks correctly is described here: https://docs.readthedocs.io/en/latest/pull-requests.html

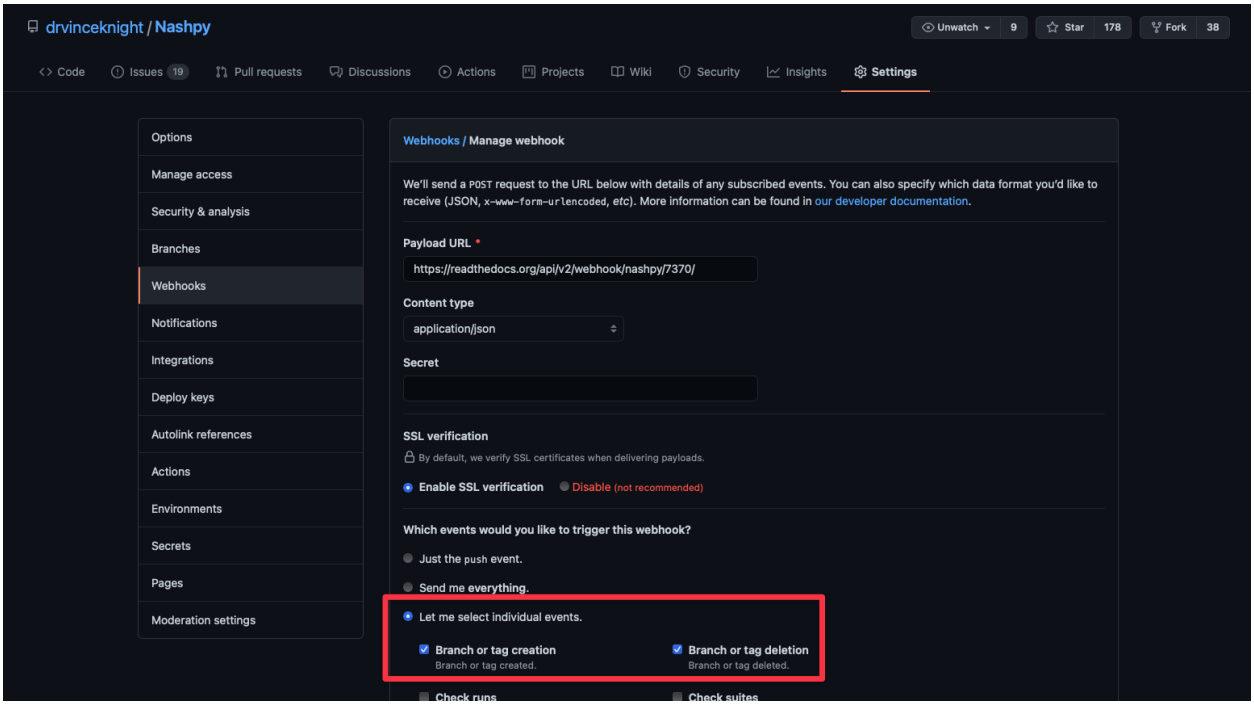
When done correctly this is what Applications settings should look like:

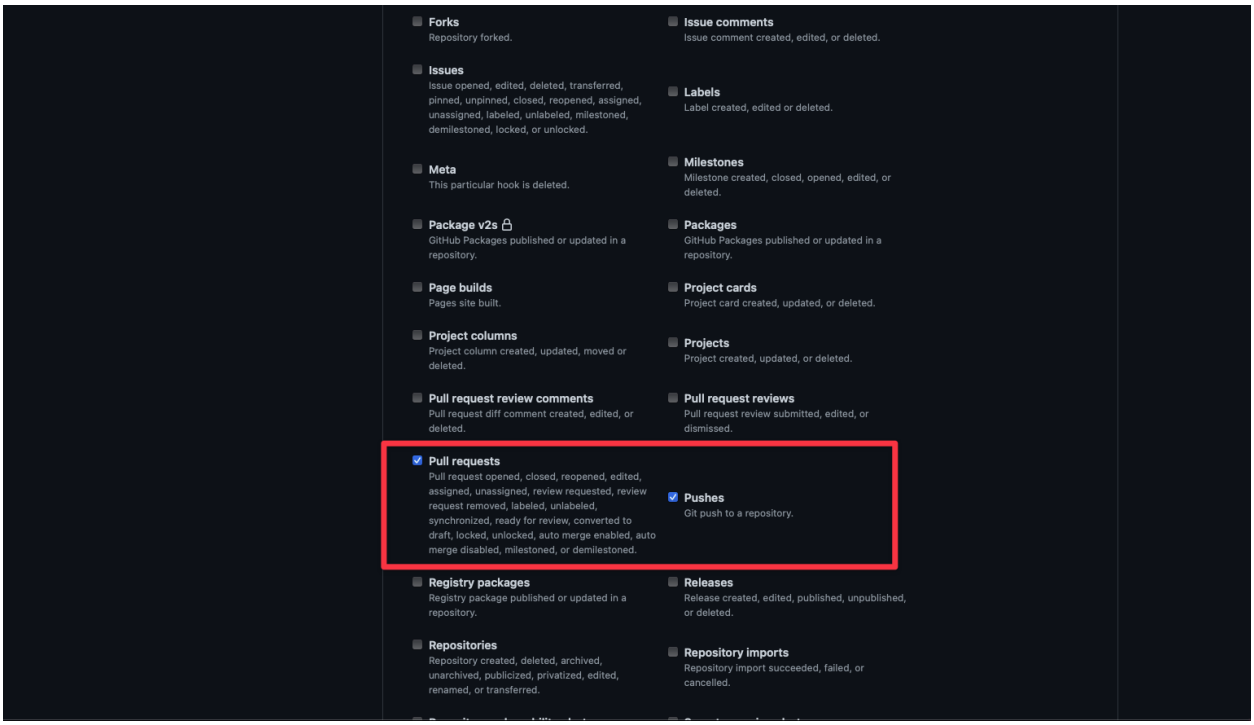


The final thing to check is the setting on the specific Github repository (under `Settings/Webhooks`) which should have the following 4 boxes ticked:

- Branch or tag creation
- Branch or tag deletion
- Pull requests
- Pushes

When done correctly this should look like:

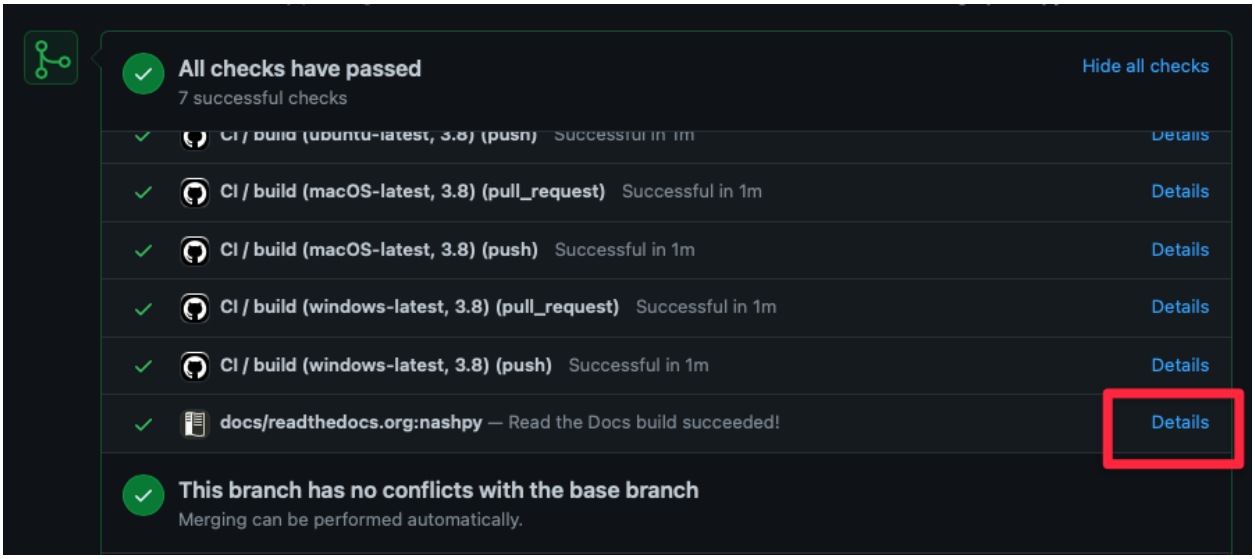




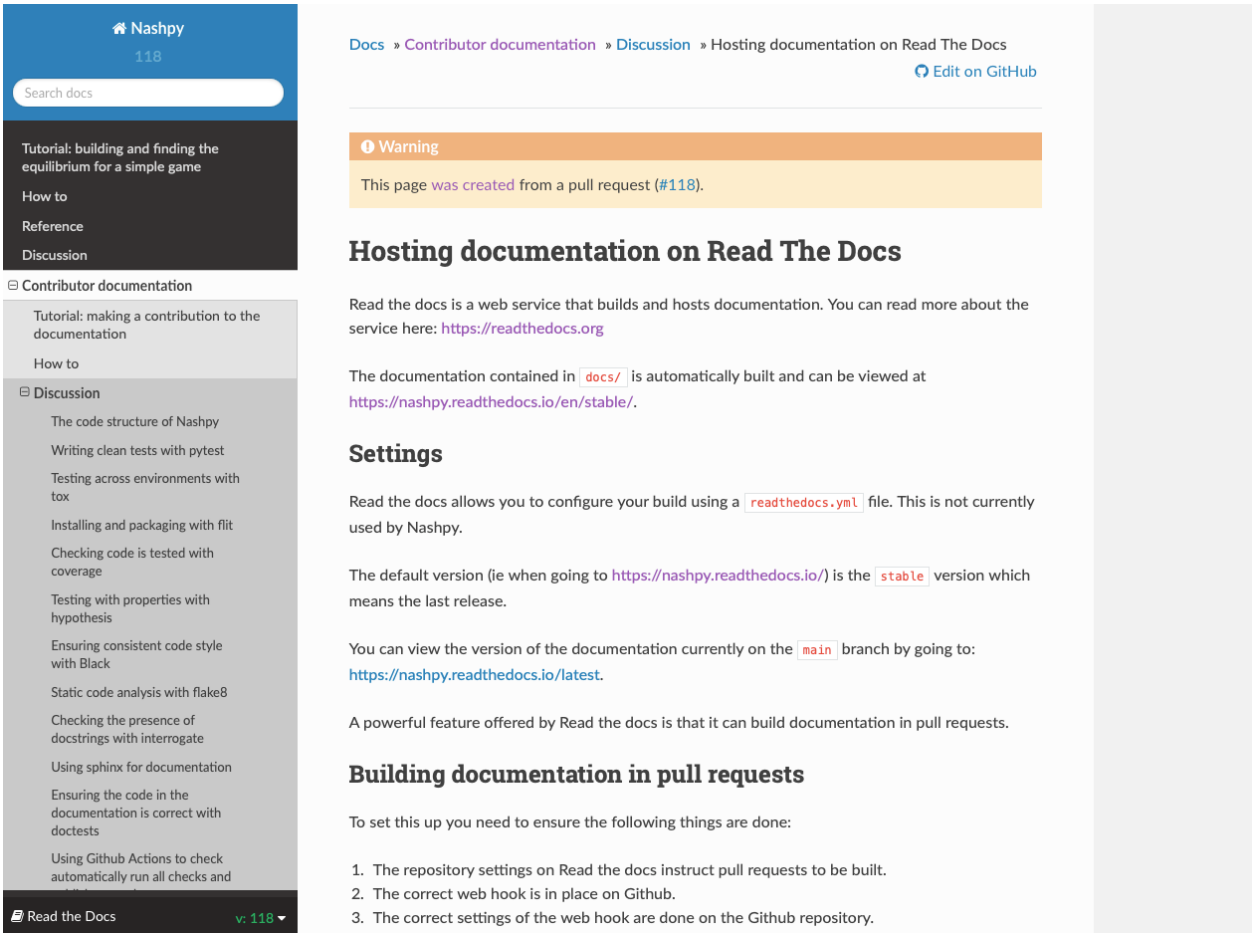
This is described here: https://docs.readthedocs.io/en/latest/webhooks.html#github (although note that ticking the Pull Requests box is not indicated there).

Reviewing documentation on Pull Requests

If this is all done correctly you will be able to view your documentation during pull requests:



For example here is how the documentation looked for pull request that added this specific page of the documentation:



5.3.18 Ensuring consistent markdown format with mdformat

From the `mdformat`'s documentation page:

> "Mdformat is an opinionated Markdown formatter that can be used to enforce a consistent style in Markdown files. Mdformat is a Unix-style command-line tool as well as a Python library."

The rationale for using `mdformat` is the same as the one for using *black* which is to avoid spending any time thinking about the formatting of source files.

Examples of choices that `mdformat` will make:

```
# Hello world

Here is how to write some python code:

   print("Hello Nashpy!)
```

become:

```
# Hello world

Here is how to write some python code:
```

(continues on next page)

(continued from previous page)

```
```
print("Hello Nashpy!)
```
```

5.4 Reference

5.4.1 Contributing Bibliography

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

5.4.2 List of contributors

- @drvinceknight
- @11michalis11
- @asinghgaba
- @katiemcgoldrick
- @volume-on-max

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Axelrod1980] Axelrod, Robert. “Effective choice in the prisoner’s dilemma.” Journal of conflict resolution 24.1 (1980): 3-25.

[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.

[Harper2017] Harper, Marc, et al. “Reinforcement learning produces dominant strategies for the iterated prisoner’s dilemma.” PloS one 12.12 (2017): e0188046.

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

[Knight2016] Knight, Vincent Anthony, et al. “An open framework for the reproducible study of the iterated prisoner’s dilemma.” Journal of Open Research Software 4.1 (2016).

[Knight2018] Knight, Vincent, et al. “Evolution reinforces cooperation with the emergence of self-recognition mechanisms: An empirical study of strategies in the Moran process for the iterated prisoner’s dilemma.” PloS one 13.10 (2018): e0204981.

[Komarova2004] Komarova, Natalia L. “Replicator–mutator equation, universality property and population dynamics of learning.” Journal of theoretical biology 230.2 (2004): 227-239.

[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).

[Maynard1974] Smith, J. Maynard. “The theory of games and the evolution of animal conflicts.” Journal of theoretical biology 47.1 (1974): 209-221.

[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.

[Moran1958] Moran, Patrick Alfred Pierce. “Random processes in genetics.” Mathematical proceedings of the cambridge philosophical society. Vol. 54. No. 1. Cambridge University Press, 1958.

[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.

[Nowak2006] Nowak, Martin A. Evolutionary dynamics: exploring the equations of life. Harvard university press, 2006.

[Press2012] Press, William H., and Freeman J. Dyson. "Iterated Prisoner's Dilemma contains strategies that dominate any evolutionary opponent." Proceedings of the National Academy of Sciences 109.26 (2012): 10409-10413.

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

[Webb2007] Webb, James N. Game theory: decisions, interaction and Evolution. Springer Science & Business Media, 2007.

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

[Procida2021] Daniele Procida. Diátaxis: A Systematic Framework For Technical Documentation Authoring. https://diataxis.fr

[Knight2018] Knight and Campbell, (2018). Nashpy: A Python library for the computation of Nash equilibria. Journal of Open Source Software, 3(30), 904, https://doi.org/10.21105/joss.00904

PYTHON MODULE INDEX

n

nashpy, 93
nashpy.algorithms.lemke_howson, 87
nashpy.algorithms.support_enumeration, 84
nashpy.algorithms.vertex_enumeration, 87
nashpy.game, 89
nashpy.learning.fictitious_play, 88

INDEX

A

asymmetric_replicator_dynamics() (*nashpy.game.Game method*), 89

F

fictitious_play() (*in module nashpy.learning.fictitious_play*), 88
fictitious_play() (*nashpy.game.Game method*), 90
fixation_probabilities() (*nashpy.game.Game method*), 90

G

Game (*class in nashpy.game*), 89
get_best_response_to_play_count() (*in module nashpy.learning.fictitious_play*), 88

I

indifference_strategies() (*in module nashpy.algorithms.support_enumeration*), 84
is_best_response() (*nashpy.game.Game method*), 90
is_ne() (*in module nashpy.algorithms.support_enumeration*), 85

L

lemke_howson() (*in module nashpy.algorithms.lemke_howson*), 87
lemke_howson() (*nashpy.game.Game method*), 91
lemke_howson_enumeration() (*nashpy.game.Game method*), 91

M

module
    nashpy, 93
    nashpy.algorithms.lemke_howson, 87
    nashpy.algorithms.support_enumeration, 84
    nashpy.algorithms.vertex_enumeration, 87
    nashpy.game, 89
    nashpy.learning.fictitious_play, 88
moran_process() (*nashpy.game.Game method*), 91

N

nashpy
    module, 93
nashpy.algorithms.lemke_howson
    module, 87
nashpy.algorithms.support_enumeration
    module, 84
nashpy.algorithms.vertex_enumeration
    module, 87
nashpy.game
    module, 89
nashpy.learning.fictitious_play
    module, 88

O

obey_support() (*in module nashpy.algorithms.support_enumeration*), 85

P

potential_support_pairs() (*in module nashpy.algorithms.support_enumeration*), 85
powerset() (*in module nashpy.algorithms.support_enumeration*), 85

R

replicator_dynamics() (*nashpy.game.Game method*), 91

S

shift_tableau() (*in module nashpy.algorithms.lemke_howson*), 88
solve_indifference() (*in module nashpy.algorithms.support_enumeration*), 86
stochastic_fictitious_play() (*nashpy.game.Game method*), 92
support_enumeration() (*in module nashpy.algorithms.support_enumeration*), 86

support_enumeration() (*nashpy.game.Game method*), 92

T

tableau_to_strategy() (*in module nashpy.algorithms.lemke_howson*), 88

U

update_play_count() (*in module nashpy.learning.fictitious_play*), 89

V

vertex_enumeration() (*in module nashpy.algorithms.vertex_enumeration*), 87
vertex_enumeration() (*nashpy.game.Game method*), 92