evoke

Release 0.1.2

Manolo Martínez & Stephen Mann

Mar 14, 2024

CONTENTS:

evoke is a Python library for evolutionary simulations of signalling games. It is particularly oriented towards reproducing results and figures from the literature, and offers a simple and intuitive API.

- Tutorial | *Google Colab*
- Documentation **You are here!** | *ReadTheDocs*
- Package | *PyPI*
- Source code | *GitHub*

Note: This project is under active development.

CHAPTER

ONE

USAGE

1.1 Installation

To use evoke, first install it using pip:

```
(.venv) $ pip install evoke_signals
```

1.2 Tutorial

The best way to discover evoke is the interactive tutorial. The rest of this document describes very simple use cases.

1.3 The simplest case: reproducing a figure from the literature

The easiest thing to do with evoke is recreate a figure from the signalling game literature. The parameters required to create some of these figures are included in the `examples` folder.

For example, to create figure 5.2 from *Signals* (Skyrms 2010), you would run:

```
Skyrms2010_5_2()
```

This creates an **object** which runs an evolutionary simulation with parameters as close as possible to those described by Bryan Skyrms for the figure in the book.

In this case evoke creates a figure very close to the original. In other cases there might be a range of random properties that can cause deviation from the figures in the literature. It's often a good idea to create the same figure multiple times, to get a feel for the range of variation that can be produced by the reported parameters.

1.4 Creating simulations

If you want to create custom simulations, you need to create two kinds of object:

- **game**: This describes properties of the game, including the probabilities of each observable state, the number of signals available, and the payoff matrices of sender and receiver.
- **evolve**: This describes properties of the evolutionary scenario, especially whether the 'agents' are populations evolving via selection or individuals learning via reinforcement.

Games and evolve objects can be mixed and matched. This allows you to see differences between evolution and reinforcement learning, by taking the same game and plugging it into different evolve objects.

CHAPTER

TWO

EXAMPLES

2.1 Skyrms (2010) *Signals*

evoke library examples from:

Skyrms, B. (2010). Signals: Evolution, Learning, and Information. Oxford University Press.

class skyrms2010signals.**Skyrms2010_1_1**

Figure 1.1, page 11, of Skyrms (2010). The figure depicts the replicator dynamics of a population repeatedly playing a two-player cooperative game where each agent in the population either always plays sender or always plays receiver. Senders observe one of two randomly chosen states of the world and produce one of two signals. Receivers observe the signal and produce one of two acts. If the act matches the state, both players gain a payoff.

initialize_simulation() → None

Sets the figure parameters as class attributes.

run_simulation()

Technically this figure does not require a simulation. It describes how a population of two types would evolve if it were playing a cooperative game.

Returns
evo – The evolutionary scenario is represented by an object from the module evolve.

Return type
an instance of evolve.TwoPops.

class skyrms2010signals.**Skyrms2010_1_2**

Figure 1.2, page 12, of Skyrms (2010). The figure depicts the replicator dynamics of a population repeatedly playing a two-player cooperative game where each agent in the population can switch between playing sender or receiver.

initialize_simulation() → None

Sets the figure parameters as class attributes.

run_simulation()

Technically this figure does not require a simulation. It describes how a population of one type would evolve if it were playing a cooperative game.

Returns
evo – The evolutionary scenario is represented by an object from the module evolve.

Return type
an instance of evolve.OnePop.

class skyrms2010signals.**Skyrms2010_3_3**(*iterations=100*)

Figure 3.3, page 40, of Skyrms (2010). The figure depicts the mutual information between signal and state over successive trials of a two-player cooperative game in which agents learn via reinforcement. Typically the mutual information will increase over time as the agents learn to use specific signals as indicators of specific states. However, the stochastic nature of the simulation means that the figure will look different each time it is run.

initialize_simulation() → None

Sets the figure parameters as class attributes.

run_simulation(*iterations*)

Create a game object and an evolution object, and run the game <iterations> times.

Parameters
iterations (*int*) – Number of timesteps in the simulation i.e. number of repetitions of the game.

Returns
evo – The evolve object controls simulations.

Return type
evolve.MatchingSR

class skyrms2010signals.**Skyrms2010_3_4**(*iterations=100*)

Figure 3.4, page 46, of Skyrms (2010). The figure depicts the change over time of the average probability of success in a cooperative signalling chain game. In this game there is a sender, an intermediary, and a receiver. There are two signals, between the sender and intermediary and between the intermediary and receiver. It takes a lot longer for signalling to become established in this game. The original figure uses one million iterations (1e6); however, the probability of success often reaches 1 after just one hundred thousand iterations (1e5).

initialize_simulation() → None

Sets the figure parameters as class attributes.

run_simulation(*iterations*)

Create a game object and an evolution object, and run the game <iterations> times.

Parameters
iterations (*int*) – Number of timesteps in the simulation i.e. number of repetitions of the game.

Returns
evo – The evolve object controls simulations.

Return type
evolve.MatchingSIR

class skyrms2010signals.**Skyrms2010_4_1**

Figure 4.1, page 59, of Skyrms (2010). The figure depicts cycles in the replicator dynamics of a rock-paper-scissors game.

initialize_simulation() → None

Sets the figure parameters as class attributes.

run_orbits()

Generate the cycles in the replicator dynamics.

The cycles are stored as class attribute self.xyzs. This is used by the superclass Ternary in the method self.show().

Returns
evo – The evolve object for a one-population game.

Return type
evolve.OnePop

class skyrms2010signals.**Skyrms2010_5_2**

Figure 5.2, page 72, of Skyrms (2010). The figure shows the value of assortment that is required to destabilise pooling in a 2x2x2 cooperative signalling game. *Assortment* is defined as the probability of meeting another player in the population who is the same type as you. *Pooling* is any strategy that produces the same signal for more than one state.

Skyrms describes this model on page 71:

"Here we consider a one-population model, in which nature assigns roles of sender or receiver on flip of a fair coin. We focus on four strategies, written as a vector whose components are: signal sent in state 1, signal sent in state 2, act done after signal 1, act done after signal 2.

s1 = <1, 2, 1, 2>

s2 = <2, 1, 2, 1>

s3 = <1, 1, 2, 2>

s4 = <2, 2, 2, 2>"

show()

Show the figure.

We call the superclass method and tell it to show the line along with the datapoints.

initialize_simulations() → None

Sets the figure parameters as class attributes.

run_simulations()

Create games and run simulations.

The vectors s1-s4 defined on page 71 of Skyrms (2010) define the playertypes payoffs.

Returns
y_axis – Results; the required assortment level for each value of pr_state_2.

Return type
list

class skyrms2010signals.**Skyrms2010_8_1**(*iterations=1000*)

Figure 8.1, page 95, of Skyrms (2010). The figure depicts the change over time of the probability of success in a two-player cooperative game where the agents learn by reinforcement.

show()

Show the figure.

We call the superclass method and tell it to show the line along with the datapoints.

Return type
None.

initialize_simulation() → None

Sets the figure parameters as class attributes.

run_simulation(*iterations*)

Create game and run simulation.

Parameters
iterations (*int*) – Number of timesteps.

Returns
evo – The simulation object.

Return type
evolve.MatchingSR

class skyrms2010signals.**Skyrms2010_8_2**(*trials=100*, *iterations=1000*)

Figure 8.2, page 97, of Skyrms (2010). The figure depicts the probability of pooling in a signalling game with reinforcement learning for different initial weights. Initial weights determine how difficult it is to learn something new: large initial weights mean that learning is slower.

Skyrms does not explicitly state the number of trials or number of iterations used to generate his figure. We suspect the parameter values are something like *trials=int(1e3)* and *iterations=int(1e5)*. However, attempting to generate this figure with those values will take an exceptionally long time.

Even with *iterations=int(1e4)*, it's looking like 12 minutes per weight, so about an hour overall.

This, combined with the difficulty of figuring out exactly how Skyrms is identifying pooling equilibria, leads to us overestimating the probability of pooling. You are warned!

show() → None

Show the figure.

We call the superclass method and tell it to show the line along with the datapoints.

initialize_simulation() → None

Sets the figure parameters as class attributes.

run_simulation(*trials*, *iterations*) → None

Create game and run simulations.

Parameters

- **trials** (*int, optional*) – Number of times to repeat a simulation with specific parameters. The default is 100.
- **iterations** (*int, optional*) – Number of timesteps in each simulation. The default is int(1e3).

class skyrms2010signals.**Skyrms2010_8_3**(*trials=100*, *iterations=300*, *learning_params=[0.01, 0.03, 0.05, 0.07, 0.1, 0.15, 0.2]*)

Figure 8.3, page 98, of Skyrms (2010). The figure depicts the probability of signalling for different values of the learning parameter in a Bush–Mosteller reinforcement scenario.

Our recreation of this figure is clearly undercounting signalling. That's because we are defining signalling as "not pooling", and we are overcounting pooling (see the docstring for class Skyrms2010_8_2).

In future, we need to try and count both pooling and signaling more accurately; this is difficult, since we don't know exactly how Skyrms defines them for the purposes of his figures.

initialize_simulation(*learning_params*) → None

Sets the figure parameters as class attributes.

Parameters
learning_params (*array-like, optional*) – Learning parameters to run simulations for.

run_simulation(*trials*, *iterations*) → None

Create game and run simulations.

Parameters

- **trials** (*int, optional*) – Number of times to repeat each simulation.
- **iterations** (*int, optional*) – Number of timesteps per simulation.

Returns
evo – Simulation object.

Return type
evolve.BushMostellerSR

class skyrms2010signals.**Skyrms2010_10_5**(*trials=1000*, *iterations=10000*)

Figure 10.5, page 130, of Skyrms (2010). The figure depicts the number of signals at the end of reinforcement for a cooperative game in which senders can invent new signals.

NB Skyrms uses trials=1000 and iterations=int(1e5) but this will take a very long time.

initialize_simulation() → None

Sets the figure parameters as class attributes.

run_simulation(*trials*, *iterations*) → None

Run <trials> trials with <iterations> iterations each.

Parameters

- **trials** (*int*) – Number of simulations.
- **iterations** (*int*) – Number of iterations per trial.

2.2 Godfrey-Smith & Martínez (2013)

evoke library examples from:

> Godfrey-Smith, P., & Martínez, M. (2013). Communication and Common Interest. *PLOS Computational Biology*, 9(11), e1003282. https://doi.org/10.1371/journal.pcbi.1003282

The supporting information (including important definitions) can be found at https://doi.org/10.1371/journal.pcbi.1003282.s001

2.2.1 How to use this script

Quick run: Create figure objects with *demo=True*.

Figures 1 and 2, full run, minimal parameters:

1. Decide how many games per value of C you want to analyse, *games_per_c*. Godfrey-Smith and Martínez use 1500.
2. Run *find_games_3x3(games_per_c)*. This will generate *games_per_c* games per value of C and store them in a local directory.
3. Run *analyse_games_3x3(games_per_c)*. This will calculate values required to create figures 1 and 2. This can take a long time! 1500 games takes about 30 minutes.
4. Run *GodfreySmith2013_1(games_per_c, demo=False)* to create Figure 1.
5. Run *GodfreySmith2013_2(games_per_c, demo=False)* to create Figure 2.

Figure 3a (sender), full run, minimal parameters:

1. Decide how many games per value of C and K you want to analyse, *games_per_c_and_k*. Godfrey-Smith and Martínez use 1500.
2. Run *find_games_3x3_c_and_k(games_per_c_and_k,sender=True)*. This will generate *games_per_c_and_k* games per pair of values C and K and store them locally.
3. Run *analyse_games_3x3_c_and_k(games_per_c_and_k,sender=True)*. This will calculate values required to create figure 3. This can take a long time!
4. Run *GodfreySmith2013_3_sender(games_per_c_and_k,demo=False)* to create Figure 3a.

Figure 3b (receiver), full run, minimal parameters:

1. Decide how many games per value of C and K you want to analyse, *games_per_c_and_k*. Godfrey-Smith and Martínez use 1500.
2. Run *find_games_3x3_c_and_k(games_per_c_and_k,sender=False)*. This will generate *games_per_c_and_k* games per pair of values C and K and store them locally.
3. Run *analyse_games_3x3_c_and_k(games_per_c_and_k,sender=False)*. This will calculate values required to create figure 3. This can take a long time!
4. Run *GodfreySmith2013_3_receiver(games_per_c_and_k,demo=False)* to create Figure 3a.

class godfreysmith2013communication.**GodfreySmith2013_1**(*games_per_c=50*, *demo=True*, *dir_games='../data/'*)

Original figure: https://doi.org/10.1371/journal.pcbi.1003282.g001

How probable is an information-using equilibrium in a randomly-chosen game with a particular level of common interest?

Common interest here is defined as Godfrey-Smith and Martínez's measure C.

2.2.2 How to use this class

You have two options to create this figure: demo mode and full mode.

- Demo mode omits hard-to-find classes and places an upper limit on *games_per_c*. This allows it to run in a reasonable amount of time.
- Full mode requires an existing set of game data stored in JSON files. These can be created via the functions *find_games_3x3()* and *analyse_games_3x3()*.

The reason for demo mode is that the figure takes a VERY long time to create with the published parameter of *games_per_c=1500*. Realistically we need to prepare by finding *games_per_c* games for each value of C, storing them in a local JSON file, and calling them at runtime to count the equilibria. Demo mode omits games with *c=0.000* and *c=0.111* because they are especially hard to find.

load_saved_games(*dir_games*, *games_per_c*) → None

Get sender and receiver matrices and load them into game objects. Put them into dictionary self.games.

The games should already exist in dir_games with filenames of the form:

f"{dir_games}games_c{c_value:.3f}_n{games_per_c}.json"

Parameters

- **dir_games** (*str*) – Directory containing JSON files with sender and receiver matrices.
- **games_per_c** (*int*) – Number of games per value of C.

create_games_demo(*games_per_c*) → None

Create game objects in demo mode.

Put them into dictionary self.games.

Parameters
games_per_c (*int*) – Number of games per value of C.

calculate_results_per_c() → None

For each value of <self.c_values>, count how many games have info-using equilibria.

class godfreysmith2013communication.**GodfreySmith2013_2**(*games_per_c=50*, *demo=True*, *dir_games='../data/'*)

Original figure: https://doi.org/10.1371/journal.pcbi.1003282.g002

What is the highest level of information transmission at equilibrium across a sample of games with a particular level of common interest?

Common interest here is defined as Godfrey-Smith and Martínez's measure C.

2.2.3 How to use this class

You have two options to create this figure: demo mode and full mode.

- Demo mode omits hard-to-find classes and places an upper limit on *games_per_c*. This allows it to run in a reasonable amount of time.
- Full mode requires an existing set of game data stored in JSON files. These can be created via the functions *find_games_3x3()* and *analyse_games_3x3()*.

The reason for demo mode is that the figure takes a VERY long time to create with the published parameter of *games_per_c=1500*. Realistically we need to prepare by finding *games_per_c* games for each value of C, storing them in a local JSON file, and calling them at runtime to count the equilibria. Demo mode omits games with *c=0.000* and *c=0.111* because they are especially hard to find.

load_saved_games(*dir_games*, *games_per_c*) → None

Get sender and receiver matrices and load them into game objects. Put them into dictionary self.games.

The games should already exist in dir_games with filenames of the form:

f"{dir_games}games_c{c_value:.3f}_n{games_per_c}.json"

Parameters

- **dir_games** (*str*) – Directory containing JSON files with sender and receiver matrices.
- **games_per_c** (*int*) – Number of games per value of C.

create_games_demo(*games_per_c*) → None

Create game objects in demo mode.

Put them into dictionary self.games.

Parameters
games_per_c (*int*) – Number of games per value of C.

calculate_results_per_c(*games_per_c*) → None

For each value of <self.c_values>, count how many out of <games_per_c> games have info-using equilibria.

Parameters
games_per_c (*int*) – Number of games to generate per level of common interest.

class godfreysmith2013communication.**GodfreySmith2013_3**(*games_per_c_and_k=150*, *k_indicator=None*, *demo=False*, *dir_games='../data/'*)

See figure at https://doi.org/10.1371/journal.pcbi.1003282.g003

This object requires an existing set of game data stored in JSON files. These can be created with *find_games_3x3_c_and_k()* and *analyse_games_3x3_c_and_k()*. See the section **How to use this script** for more.

Demo mode is not yet available for this figure.

load_saved_games(*dir_games*) → None

Get sender and receiver matrices and load them into game objects. Put them into dictionary self.games.

The games should already exist in dir_games with filenames of the form:

f"{dir_games}games_c{c_value:.3f}_{ks or kr}{k_value:.3f}_n{games_per_c_and_k}.json"

Parameters
dir_games (*str*) – Directory containing JSON files with sender and receiver matrices.

calculate_results_per_c_and_k() → None

For each pair of *self.c_values* and *self.k_values*, count how many games have info-using equilibria.

class godfreysmith2013communication.**GodfreySmith2013_3_sender**(***kwargs*)

Wrapper for GodfreySmith2013_3(), calling with parameter *self.k_indicator = "ks"* to create figure 3a.

class godfreysmith2013communication.**GodfreySmith2013_3_receiver**(***kwargs*)

Wrapper for GodfreySmith2013_3(), calling with parameter *self.k_indicator = "kr"* to create figure 3b.

godfreysmith2013communication.**calculate_D**(*payoff_matrix*, *state*, *act_1*, *act_2*) → float

Calculate an agent's relative preference of acts *act_1* and *act_2* in state *state*.

The measure is defined in the supplement of Godfrey-Smith and Martínez (2013), page 1.

Parameters

- **payoff_matrix** (*array-like*) – The agent's payoff matrix.
- **state** (*int*) – Index of the state.
- **act_1** (*int*) – Index of the first act to be compared.
- **act_2** (*int*) – Index of the second act to be compared.

Returns

D – Godfrey-Smith and Martínez's measure D.

- 0 if act 1 is preferred
- 0.5 if the payoffs are equal
- 1 if act 2 is preferred

Return type
float

godfreysmith2013communication.**calculate_C**(*state_chances*, *sender_payoff_matrix*, *receiver_payoff_matrix*) → float

Calculate C as per Godfrey-Smith and Martínez's definition.

See page 2 of the supporting information at https://doi.org/10.1371/journal.pcbi.1003282.s001

Returns
c – PGS & MM's measure C.

Return type
float

godfreysmith2013communication.**calculate_Ks_and_Kr**(*sender_payoff_matrix*, *receiver_payoff_matrix*)

Calculate the extent to which an agent's preference ordering over receiver actions varies with the state of the world.

Defined as *K_S* and *K_R* in the supplement of Godfrey-Smith and Martínez (2013), page 2.

Parameters
payoff_matrix (*array-like*) – The agent's payoff matrix.

Returns
K

Return type
float

godfreysmith2013communication.**find_games_3x3**(*games_per_c=1500*, *c_values=array([0., 0.1111, 0.2222, 0.3333, 0.4444, 0.5556, 0.6667, 0.7778, 0.8889, 1.])*, *dir_games='../data/'*) → None

Finds *games_per_c* 3x3 sender and receiver matrices and saves them as JSON files, storing them by C value in *dir_games*.

Since it's hard to find games for certain values of C, we'll save each JSON file individually once we've found it. Then if you have to terminate early, you can come back and just search for games with the values of C you need later on.

Parameters

- **dir_games** (*str*) – Directory to place JSON files
- **games_per_c** (*int, optional*) – Number of games to find per value of c. The default is 1500.
- **c_values** (*array-like*) – List of C values to find games for. The default is the global variable c_3x3_equiprobable.

Return type
None.

godfreysmith2013communication.**analyse_games_3x3**(*games_per_c=1500*, *c_values=array([0., 0.1111, 0.2222, 0.3333, 0.4444, 0.5556, 0.6667, 0.7778, 0.8889, 1.])*, *dir_games='../data/'*, *sigfig=5*) → None

Find information-using equilibria of 3x3 games and the mutual information between states and acts at those equilibria.

The games should already exist in *dir_games* with filenames of the form:

```
f"{dir_games}games_c{c_value:.3f}_n{games_per_c}.json"
```

Each file should be a list of dicts. Each dict corresponds to a game:

```
{ "s": <sender payoff matrix> "r": <receiver payoff matrix> "e": <equilibrium
with the highest information transmission> "i": <mutual information between states
and acts at this equilibrium> }
```

`s` and `r` already exist; this function fills in `e` and `i`.

Parameters

- **dir_games** (*str*) – Directory to find and store JSON files
- **games_per_c** (*int, optional*) – Number of games to find per value of c. The default is 1500.
- **c_values** (*array-like*) – List of C values to find games for. The default is the global variable c_3x3_equiprobable.
- **sigfig** (*int, optional.*) – The number of significant figures to report values in. Since gambit sometimes has problems rounding, it generates values like 0.9999999999996. We want to report these as 1.0000, especially if we're dumping to a file. The default is 5.

godfreysmith2013communication.**find_games_3x3_c_and_k**(*games_per_c_and_k=1500*, *sender=True*, *c_values=array([0., 0.1111, 0.2222, 0.3333, 0.4444, 0.5556, 0.6667, 0.7778, 0.8889, 1.])*, *k_values=array([0., 0.3333, 0.6667, 1., 1.3333, 1.6667, 2.])*, *dir_games='../data/'*) → None

Finds games_per_c_and_k 3x3 sender and receiver matrices and saves them as JSON files, storing them by C and K values in dir_games.

Note that it is EXTREMELY difficult to find games for some combinations of C and K, especially when C=0. Expect this to take a long time!

Since it's hard to find games for certain combinations of C and K, we'll save each JSON file individually once we've found it. Then if you have to terminate early, you can come back and just search for games with the combinations of C and K you need later on.

Parameters

- **games_per_c_and_k** (*int, optional*) – Number of games to find per pair of c and k. The default is 1500.
- **sender** (*bool*) – If True, the operative value of K is the sender's K_S If False, the operative value of K is the receiver's K_R
- **c_values** (*array-like*) – List of C values to find games for. The default is the global variable c_3x3_equiprobable.
- **k_values** (*array-like*) – List of K values to find games for. The default is the global variable k_3x3.
- **dir_games** (*str*) – Directory to place JSON files

godfreysmith2013communication.**analyse_games_3x3_c_and_k**(*games_per_c_and_k=1500*, *sender=True*, *c_values=array([0., 0.1111, 0.2222, 0.3333, 0.4444, 0.5556, 0.6667, 0.7778, 0.8889, 1.])*, *k_values=array([0., 0.3333, 0.6667, 1., 1.3333, 1.6667, 2.])*, *dir_games='../data/'*, *sigfig=5*) → None

Find information-using equilibria of 3x3 games and the mutual information between states and acts at those equilibria.

The games should already exist in dir_games with filenames of the form:

```
f"{dir_games}games_c{c_value:.3f}_{ks or kr}{k_value:.3f}_n{games_per_c}.json"
```

Each file should be a list of dicts. Each dict corresponds to a game:

```
{ "s": <sender payoff matrix> "r": <receiver payoff matrix> "e": <equilibrium
with the highest information transmission> "i": <mutual information between states
and acts at this equilibrium> }
```

s and r already exist; this function fills in e and i.

Parameters

- **games_per_c_and_k** (*int, optional*) – Number of games to analyse per pair of c and k. The default is 1500.
- **sender** (*bool*) – If True, the operative value of K is the sender's K_S If False, the operative value of K is the receiver's K_R
- **c_values** (*array-like*) – List of C values to analyse games for. The default is the global variable c_3x3_equiprobable.
- **k_values** (*array-like*) – List of K values to analyse games for. The default is the global variable k_3x3.
- **dir_games** (*str*) – Directory to find and update JSON files
- **sigfig** (*int, optional.*) – The number of significant figures to report values in. Since gambit sometimes has problems rounding, it generates values like 0.9999999999996. We want to report these as 1.0000, especially if we're dumping to a file. The default is 5.

godfreysmith2013communication.**get_random_payoffs**(*states=3*, *acts=3*, *min_payoff=0*, *max_payoff=100*)

Generate a random payoff matrix.

Parameters

- **states** (*int, optional*) – Number of states observable by the sender. The default is 3.
- **acts** (*int, optional.*) – Number of acts available to the receiver.
- **min_payoff** (*int, optional*) – Smallest possible payoff. The default is 0.
- **max_payoff** (*int, optional*) – Largest possible payoff. The default is 100.

Returns

payoffs – A random payoff matrix of shape (states,acts).

Return type

array-like

CHAPTER

THREE

FULL API

3.1 Figure objects `figure.py`

Figure objects.

Figures can be plotted by calling an instance of the relevant class,
e.g. f = Skyrms2010_3_3() will create a figure object and simultaneously plot it.

`class figure.Figure(evo=None, game=None, **kwargs)`

Abstract superclass for all figures.

`abstract show()`

Show the plot of the figure with the current parameters, typically with plt.show().

This is an abstract method that must be redefined for each subclass.

Return type
None.

`abstract reset()`

Reset parameters for the figure.

This is an abstract method that must be redefined for each subclass.

Return type
None.

`classmethod demo_warning()`

Warn the user that they are running in demo mode.

Return type
None.

`property properties`

Get a dict of all the editable properties of the figure, with their current values.

Typically includes properties like plot color, axis labels, plot scale etc.

Returns
list_of_properties – A list of the editable properties of the object.

Return type
list

`class figure.Scatter(**kwargs)`

Superclass for scatter plots

reset(*x*, *y*, *xlabel*, *ylabel*, *marker_size=10*, *marker_color='k'*, *xlim=None*, *ylim=None*, *xscale=None*, *yscale=None*)

Update figure parameters, which can then be plotted with self.show().

Parameters

- **x** (*array-like*) – x-axis coordinates.
- **y** (*array-like*) – y-axis coordinates.
- **xlabel** (*str*) – x-axis label.
- **ylabel** (*str*) – y-axis label.
- **marker_size** (*int, optional*) – Size of the markers for each data point. The default is 10.
- **marker_color** (*str, optional*) – Color of the datapoint markers. The default is "k".
- **xlim** (*array-like, optional*) – Minimum and maximum values of x-axis. The default is None.
- **ylim** (*array-like, optional*) – Minimum and maximum values of y-axis. The default is None.
- **xscale** (*str, optional*) – x-axis scaling i.e. linear or logarithmic. The default is None.
- **yscale** (*str, optional*) – y-axis scaling i.e. linear or logarithmic. The default is None.

Return type
None.

show()

Show figure with the current parameters.

Parameters
line (*bool, optional*) – Whether to show a line connecting the datapoints. The default is False.

Return type
None.

property marker_size

Marker color

property marker_color

Line connecting the markers

class figure.**Quiver**(***kwargs*)

Superclass for Quiver plots

class figure.**Quiver2D**(*scale=20*, ***kwargs*)

Plot a 2D quiver plot.

reset(*color=None*, *xlabel=None*, *ylabel=None*)

Reset parameters for the figure.

This is an abstract method that must be redefined for each subclass.

Return type
None.

show()

Display the 2D quiver plot with the loaded data.

Raises
NoDataException – The requisite data has not been supplied by the user.

Return type
None.

uv_from_xy(*x*, *y*)

Parameters

- **x** (*float*) – Current proportion of the first sender strategy.
- **y** (*float*) – Current proportion of the first receiver strategy.
- **Returns** –
- **strategy.** (*velocity of SECOND receiver*) –
- **strategy.** –

class figure.**Quiver3D**(*color='k'*, *normalize=True*, *length=0.5*, *arrow_length_ratio=0.5*, *pivot='middle'*, ***kwargs*)

Plot a 3D quiver plot.

reset()

Reset parameters for the figure.

This is an abstract method that must be redefined for each subclass.

Return type
None.

show()

Display the 3D quiver plot with the loaded data.

Raises
NoDataException – The requisite data has not been supplied.

Return type
None.

vector_to_barycentric(*vector*)

Convert a 4d vector location to its equivalent within a tetrahedron

Parameters
vector (*TYPE*) – DESCRIPTION.

Returns
barycentric_location – DESCRIPTION.

Return type
TYPE

class figure.**Bar**(***kwargs*)

Bar chart abstract superclass.

reset(*x*, *y*, *xlabel*, *ylabel*, *bar_color='w'*, *xlim=None*, *ylim=None*, *yscale=None*)

Update figure parameters

Parameters

- **x** (*array-like*) – x-axis coordinates.
- **y** (*array-like*) – y-axis coordinates.

Return type
None.

show()

Display the bar chart with the loaded data.

Raises
NoDataException – The requisite data has not been supplied. Bar charts need x-axis values and y-axis values.

Return type
None.

class figure.**Ternary**(***kwargs*)

Superclass for ternary (2-simplex) plots

reset(*right_corner_label*, *top_corner_label*, *left_corner_label*, *fontsize*)

Reset parameters for the figure.

This is an abstract method that must be redefined for each subclass.

Return type
None.

show()

Display the ternary plot with the loaded data.

Raises
NoDataException – The requisite data was not supplied. Ternary plots require an <xyzs> attribute.

Return type
None.

class figure.**Surface**(***kwargs*)

Superclass for 3D surface plots (e.g. colormap). Uses ax.plot_surface().

reset(*x=None*, *y=None*, *z=None*, *xlabel=None*, *ylabel=None*, *zlabel=None*, *xlim=None*, *ylim=None*, *zlim=None*, *cmap=<matplotlib.colors.LinearSegmentedColormap object>*, *linewidth=1*, *antialiased=False*, *elev=25.0*, *azim=245*, *dist=12*) → None

Update figure parameters, which can then be plotted with self.show().

Parameters

- **x** (*array-like*) – x-axis values.
- **y** (*array-like*) – y-axis values.
- **z** (*array-like*) – z-axis values.
- **xlim** (*array-like, optional*) – Minimum and maximum values of x-axis. The default is None.
- **ylim** (*array-like, optional*) – Minimum and maximum values of y-axis. The default is None.
- **zlim** (*array-like, optional*) – Minimum and maximum values of z-axis. The default is None.

- **cmap** (*matplotlib.colors.LinearSegmentedColormap, optional*) – Color mapping. The default is cm.coolwarm.
- **linewidth** (*float(?) or int, optional*) – Width of the lines in the surface. The default is 1.
- **antialiased** (*bool, optional*) – Whether the figure is antialiased. The default is False.
- **elev** (*float*) – camera elevation
- **azim** (*int*) – camera azimuth
- **dist** (*int*) – camera distance

Return type
None.

show() → None

Show figure with current parameters.

Return type
None.

3.2 Evolve objects evolve.py

Calculate equations to evolve populations in a game. Right now, we can calculate the replicator (-mutator) dynamics, with one or two populations, in discrete or continuous time

class evolve.**OnePop**(*game*, *playertypes*)

Calculate the equations necessary to evolve one population. It takes as input a <game> and an array such that the <i,j> cell gives the expected payoff for the-strategist player of an encounter in which they follow strategy i and their opponent follows strategy j.

random_player()

Return frequencies of a random sender population

avg_payoff(*player*)

Return the average payoff that players get when the population vector is <player>

avg_payoff_vector(*player*)

Get expected payoffs of every type when population vector is <player>.

Depends on assortment.

p(s_i meets s_i) = p(s_i) + self.e * (1-p(s_i)) p(s_i meets s_j) = p(s_j) - self.e * p(s_j)

Parameters
player (*TYPE*) – DESCRIPTION.

Return type
None.

replicator_dX_dt_odeint(*X*, *t*)

Calculate the rhs of the system of odes for scipy.integrate.odeint

replicator_jacobian_odeint(*X*, *t=0*)

Calculate the Jacobian of the system of odes for scipy.integrate.odeint

discrete_replicator_delta_X(*X*)

Calculate a population vector for t' given the vector for t, using the discrete time replicator dynamics (Huttegger 2007)

replicator_odeint(*init*, *time_vector*, ***kwargs*)

Calculate one run of the game following the replicator(-mutator) dynamics, with starting points sinit and rinit, in times <times> (an evolve.Times instance), using scipy.integrate.odeint

replicator_discrete(*initpop*, *steps*)

Calculate one run of the game, following the discrete replicator(-mutator) dynamics, for <steps> steps with starting population vector <initpop> using the discrete time replicator dynamics.

pop_to_mixed_strat(*pop*)

Take a population vector and output the average strat implemented by the whole population

assortment(*e*)

Set assortment level

Parameters
e (*TYPE*) – DESCRIPTION.

Return type
None.

class evolve.**TwoPops**(*game*, *sendertypes*, *receivertypes*)

Calculate the equations necessary to evolve a population of senders and one of receivers. It takes as input a <game>, (which as of now only can be a Chance object), and a tuple: the first (second) member of the tuple is a nxm array such that the <i,j> cell gives the expected payoff for the sender (receiver) of an encounter in which the sender follows strategy i and the receiver follows strategy j.

random_sender()

Return frequencies of a random sender population

random_receiver()

Return frequencies of a random receiver population

sender_avg_payoff(*sender*, *receiver*)

Return the average payoff that senders get when the population vectors are <sender> and <receiver>

receiver_avg_payoff(*receiver*, *sender*)

Return the average payoff that receivers get when the population vectors are <sender> and <receiver>

replicator_dX_dt_odeint(*X*, *t*)

Calculate the rhs of the system of odes for scipy.integrate.odeint

replicator_dX_dt_ode(*t*, *X*)

Calculate the rhs of the system of odes for scipy.integrate.ode

replicator_jacobian_odeint(*X*, *t=0*)

Calculate the Jacobian of the system for scipy.integrate.odeint

replicator_jacobian_ode(*t*, *X*)

Calculate the Jacobian of the system for scipy.integrate.ode

discrete_replicator_delta_X(*X*)

Calculate a population vector for t' given the vector for t, using the discrete time replicator dynamics (Huttegger 2007)

replicator_odeint(*sinit*, *rinit*, *times*, ***kwargs*)

Calculate one run of the game following the replicator(-mutator) dynamics, with starting points sinit and rinit, in times <times> (an evolve.Times instance), using scipy.integrate.odeint

replicator_ode(*sinit*, *rinit*, *times*, *integrator='dopri5'*)

Calculate one run of the game, following the replicator(-mutator) dynamics in continuous time, in <times> (an evolve.Times instance) with starting points sinit and rinit using scipy.integrate.ode

replicator_discrete(*sinit*, *rinit*, *times*)

Calculate one run of the game, following the discrete replicator(-mutator) dynamics, in <times> (an evolve.Times object) with starting population vector <popvector> using the discrete time replicator dynamics. Note that this solver will just calculate n points in the evolution of the population, and will not try to match them to the times as provided.

vector_to_populations(*vector*)

Take one of the population vectors returned by the solvers, and output two vectors, for the sender and receiver populations respectively.

sender_to_mixed_strat(*senderpop*)

Take a sender population vector and output the average sender strat implemented by the whole population

receiver_to_mixed_strat(*receiverpop*)

Take a receiver population vector and output the average receiver strat implemented by the whole population

class evolve.**Reinforcement**(*game*, *agents*)

Evolving finite sets of agents by reinforcement learning.

reset()

Initialise values and agents.

Return type
None.

run(*iterations*, *hide_progress=True*, *calculate_stats='step'*)

Run the simulation for <iterations> steps.

Parameters

- **iterations** (*int*) – Number of times to call self.step().
- **hide_progress** (*bool*) – Whether to display tqdm progress bar
- **calculate_stats** (*str*) – When to calculate stats "step": every step "end": only at the last step

Return type
None.

is_pooling(*epsilon=0.001*)

Determine whether the current strategies are pooling or a signalling system.

If the mutual information between states and acts at the current population state is within <epsilon> of the maximum possible, it's a signalling system. Otherwise, it's pooling.

Clearly if the number of signals is lower than both the number of states and the number of acts, it will necessarily be pooling.

Parameters
epsilon (*float*) – How close to the maximum possible mutual information must the current mutual information be in order to count as a signalling system?

Returns
pooling – True if the current strategies constitute a pooling equilibrium.

Return type
bool

class evolve.**Matching**(*game*, *agents*)

Reinforcement according to Richard Herrnstein's matching law. The probability of choosing an action is proportional to its accumulated rewards.

class evolve.**MatchingSR**(*game*, *sender_strategies*, *receiver_strategies*)

Matching simulation for two-player sender-receiver game.

step(*calculate_stats=True*)

Implement the matching rule and increment one step.

In each step:

1. Run one round of the game.
2. Update the agents' strategies based on the payoffs they received.
3. Calculate and store any required variables e.g. information.
4. Update iteration.

Return type
None.

calculate_stats()

Calculate and store informational quantities at this point.

Return type
None.

class evolve.**MatchingSRInvention**(*game*, *sender_strategies*, *receiver_strategies*)

Matching simulation for two-player sender-receiver game,
with the possibility of new signals at every step.

step(*calculate_stats=True*)

Implement the matching rule and increment one step.

In each step:

1. Run one round of the game.
2. Update the agents' strategies based on the payoffs they received.
3. Calculate and store any required variables e.g. information.
4. Update iteration.

Return type
None.

calculate_stats()

Calculate and store informational quantities at this point.

Return type
None.

class evolve.**MatchingSIR**(*game*, *sender_strategies*, *intermediary_strategies*, *receiver_strategies*)

Reinforcement game for sender, intermediary, receiver.

step(*calculate_stats=True*)

Implement the matching rule and increment one step.

In each step:

1. Run one round of the game.
2. Update the agents' strategies based on the payoffs they received.
3. Calculate and store any required variables e.g. probability of success.
4. Update iteration.

Return type
None.

record_probability_of_success()

For now, just store "probability of success."

Return type
None.

class evolve.**BushMostellerSR**(*game*, *sender_strategies*, *receiver_strategies*, *learning_parameter*)

Bush_mosteller reinforcement simulation for two-player sender-receiver game.

step(*calculate_stats=True*)

Implement the matching rule and increment one step.

In each step:

1. Run one round of the game.
2. Update the agents' strategies based on the payoffs they received.
3. Calculate and store any required variables e.g. information.
4. Update iteration.

Return type
None.

calculate_stats()

Calculate and store informational quantities at this point.

Return type
None.

class evolve.**Agent**(*strategies*)

Finite, discrete agent used in Reinforcement() objects.

choose_strategy(*input_data*)

Sample from self.strategies[input_data] to get a concrete response.

When the strategies are simply a matrix,
with each row defining a distribution over possible responses, <input_data> is an integer indexing a row of the array.

So we choose that row and choose randomly from it,
according to the conditional probabilities of the responses, which are themselves listed as entries in each row.

E.g. if this is a sender, <input_data> is the index of the current state of the world,
and the possible responses are the possible signals.

If this is a receiver, <input_data> is the index of the signal sent,
and the possible responses are the possible acts.

Returns
response – The index of the agent's response.

Return type
int.

`update_strategies`(*input_data*, *response*, *payoff*)

The agent has just played <response> in response to <input_data>,
and received <payoff> as a result.

They now update the probability of choosing that response for
that input data, proportionally to <payoff>.

Parameters

- `input_data` (*TYPE*) – DESCRIPTION.
- `response` (*TYPE*) – DESCRIPTION.
- `payoff` (*TYPE*) – DESCRIPTION.

Return type
None.

`update_strategies_bush_mosteller`(*input_data*, *response*, *payoff*, *learning_parameter*)

From Skyrms 2010 page 86:

"If an act is chosen and a reward is gotten
the probability is incremented by adding some fraction of the distance between the original probability and probability one. Alternative action probabilities are decremented so that everything adds to one. The fraction used is the product of the reward and some learning parameter."

Parameters

- `input_data` (*TYPE*) – DESCRIPTION.
- `response` (*TYPE*) – DESCRIPTION.
- `payoff` (*TYPE*) – DESCRIPTION.
- `learning_parameter` (*TYPE*) – DESCRIPTION.

Return type
None.

`add_signal_sender`()

TODO: consolidate with add_signal_receiver(), and tell the agent who it is

Return type
None.

add_signal_receiver()

TODO: consolidate with add_signal_sender(), and tell the agent who it is

Return type
None.

class evolve.**Times**(*initial_time*, *final_time*, *time_inc*)

Provides a way of having a single time input to both odeint and ode

evolve.**mutationmatrix**(*mutation*, *dimension*)

Calculate a (square) mutation matrix with mutation rate given by <mutation> and dimension given by <dimension>

3.3 Game objects games.py

Set up an asymmetric evolutionary game, that can be then fed to the evolve module. There are two main classes here:

- Chance: Games with a chance player
- NonChance: Games without a chance player

class games.**Chance**(*state_chances*, *sender_payoff_matrix*, *receiver_payoff_matrix*, *messages*)

Construct a payoff function for a game with a chance player, that chooses a state, among m possible ones; a sender that chooses a message, among n possible ones; a receiver that chooses an act among o possible ones; and the number of messages

choose_state()

Return a random state, relying on the probabilities given by self.state_chances

sender_payoff(*state*, *act*)

Return the sender payoff for a combination of <state> and <act>

receiver_payoff(*state*, *act*)

Return the receiver payoff for a combination of <state> and <act>

sender_pure_strats()

Return the set of pure strategies available to the sender

receiver_pure_strats()

Return the set of pure strategies available to the receiver

one_pop_pure_strats()

Return the set of pure strategies available to players in a one-population setup

payoff(*sender_strat*, *receiver_strat*)

Calculate the average payoff for sender and receiver given concrete sender and receiver strats

avg_payoffs(*sender_strats*, *receiver_strats*)

Return an array with the average payoff of sender strat i against receiver strat j in position <i, j>

create_gambit_game()

Create a gambit object based on this game.

[SFM: UPDATE: this method has changed significantly to comply with pygambit 16.1.0. Original note: For guidance creating this method I followed the tutorial at https://nbviewer.org/github/gambitproject/gambit/blob/master/contrib/samples/sendrecv.ipynb and adapted as appropriate.]

Returns
g

Return type
Game() object from pygambit package.

property has_info_using_equilibrium: bool

Does this game have an information-using equilibrium?

Parameters
sigfig (*int, optional*) – The number of significant figures to report values in. Since gambit sometimes has problems rounding, it generates values like 0.9999999999996. We want to report these as 1.0000, especially if we're dumping to a file. The default is 5.

Returns
If True, the game has at least one information-using equilibrium. If False, the game does not have an information-using equilibrium.

Return type
bool

property highest_info_using_equilibrium: tuple

Get the mutual information between states and acts at the equilibrium with the highest such value. Also get the strategies at this equilibrium

Note that if the game has no information-using equilibria, the value of mutual information will be 0. The strategies returned will then be an arbitrary equilibrium.

Parameters
sigfig (*int, optional*) – The number of significant figures to report values in. Since gambit sometimes has problems rounding, it generates values like 0.9999999999996. We want to report these as 1.0000, especially if we're dumping to a file. The default is 5.

Returns
First element is a list containing the highest-info-using sender and receiver strategies. Second element is the mutual information between states and acts given these strategies.

Return type
tuple

property max_mutual_info

Maximum possible mutual information between states and acts. Depends on self.state_chances.

Lazy instantiation

Returns
_max_mutual_info – The maximum mutual information between states and acts for this game.

Return type
float

class games.**ChanceSIR**(*state_chances=array([0.5, 0.5])*, *sender_payoff_matrix=array([[1., 0.], [0., 1.]])*, *intermediary_payoff_matrix=array([[1., 0.], [0., 1.]])*, *receiver_payoff_matrix=array([[1., 0.], [0., 1.]])*, *messages_sender=2*, *messages_intermediary=2*)

A sender-intermediary-receiver game with Nature choosing the state.

choose_state()

Randomly get a state according to self.state_chances

Returns
state – Index of the chosen state.

Return type
int

`payoff_sender`(*state*, *act*)

Get the sender's payoff when this combination of state and act occurs.

Parameters

- `state` (*TYPE*) – DESCRIPTION.
- `act` (*TYPE*) – DESCRIPTION.

Returns
payoff – DESCRIPTION.

Return type
TYPE

`payoff_intermediary`(*state*, *act*)

Get the intermediary's payoff when this combination of state and act occurs.

Parameters

- `state` (*TYPE*) – DESCRIPTION.
- `act` (*TYPE*) – DESCRIPTION.

Returns
payoff – DESCRIPTION.

Return type
TYPE

`payoff_receiver`(*state*, *act*)

Get the receiver's payoff when this combination of state and act occurs.

Parameters

- `state` (*TYPE*) – DESCRIPTION.
- `act` (*TYPE*) – DESCRIPTION.

Returns
payoff – DESCRIPTION.

Return type
TYPE

`avg_payoffs_regular`(*snorm*, *inorm*, *rnorm*)

Return the average payoff of all players given these strategy profiles.

Requires game to be regular.

Parameters

- `snorm` (*array-like*) – Sender's strategy profile, normalised.
- `inorm` (*array-like*) – Intermediary player's strategy profile, normalised.
- `rnorm` (*array-like*) – Receiver's strategy profile, normalised.

Returns
payoff – The average payoff given these strategies. The payoff is the same for every player.

Return type
float

class games.**NonChance**(*sender_payoff_matrix*, *receiver_payoff_matrix*, *messages*)

Construct a payoff function for a game without chance player: a sender that chooses a message, among n possible ones; a receiver that chooses an act among o possible ones; and the number of messages

sender_pure_strats()

Return the set of pure strategies available to the sender. For this sort of games, a strategy is a tuple of vector with probability 1 for the sender's state, and an mxn matrix in which the only nonzero row is the one that correspond's to the sender's type.

receiver_pure_strats()

Return the set of pure strategies available to the receiver

payoff(*sender_strat*, *receiver_strat*)

Calculate the average payoff for sender and receiver given concrete sender and receiver strats

avg_payoffs(*sender_strats*, *receiver_strats*)

Return an array with the average payoff of sender strat i against receiver strat j in position <i, j>

create_gambit_game()

Create a gambit object based on this game.

[SFM: UPDATE: this method has changed significantly to comply with pygambit 16.1.0. Original note: For guidance creating this method I followed the tutorial at https://nbviewer.org/github/gambitproject/gambit/blob/master/contrib/samples/sendrecv.ipynb and adapted as appropriate.]

Returns
g

Return type
Game() object from pygambit package.

class games.**NoSignal**(*payoff_matrix*)

Construct a payoff function for a game without chance player: and in which no one signals. Both players have the same payoff matrix

pure_strats()

Return the set of pure strategies available to the players. For this sort of games, a strategy is a probablity vector over the set of states

payoff(*first_player*, *second_player*)

Calculate the average payoff for first and second given concrete strats

avg_payoffs(*player_strats*)

Return an array with the average payoff of strat i against strat j in position <i, j>

games.**lewis_square**(*n=2*)

Factory method to produce a cooperative nxnxn signalling game (what Skyrms calls a "Lewis signalling game").

Returns
game – A nxnxn cooperative signalling game.

Return type
Chance object.

games.**gambit_example**(*n=2*, *export=False*, *fpath='tester.efg'*)

Create the gambit representation of a cooperative nxnxn game
and compute its Nash equilibria.

Optionally output as an extensive-form game, which can be
loaded into the Gambit GUI.

Return type
None.

3.4 Calculations calculate.py

Solve large batches of games. There are a bunch of idyosincratic functions here. This module is mostly for illustration of use cases.

calculate.**one_basin_discrete**(*game*, *trials*, *times*)

Calculate evolutions for <trials> starting points of <game> (which is an instance of game.Evolve), in <times> (an instance of game.Times)

calculate.**one_basin_discrete_aux**(*triple*)

Calculate the one_basin loop using replicator_discrete

calculate.**one_basin_mixed**(*game*, *trials*, *times*)

Calculate evolutions for <trials> starting points of <game> (which is an instance of game.Evolve), in <times> (an instance of game.Times)

calculate.**one_basin_aux_mixed**(*triple*, *print_trials=True*)

Calculate the one_basin loop. First lsoda, then dopri5 if error

calculate.**one_basin_aux**(*triple*)

Calculate the one_basin loop using replicator_odeint

calculate.**one_basin_ode_aux**(*triple*)

Calculate the one_basin loop using one_run_ode

calculate.**one_batch**(*fileinput*, *directory*, *alreadydone=''*)

Take all games in <fileinput> and calculate one_basin on each. Save in <directory>

calculate.**pop_vector**(*vector*)

Test if <vector> is a population vector: sums a total of 2, and every value is larger or equal than zero

calculate.**test_endstate**(*array*)

Test if <array> is composed by two concatenated probability vectors

3.5 Calculations relating to common interest common_interest.py

Analyses of common interest

class common_interest.**CommonInterest_1_pop**(*game*)

Calculate quantities useful for the study of the degree of common interest between senders and receivers

K(*array*)

Calculate K as defined in Godfrey-Smith and Martinez (2013) – but using scipy.stats.kendalltau

sender_K()

Calculate K for the sender

receiver_K()

Calculate K for the receiver

C_chance()

Calculate C as defined in Godfrey-Smith and Martinez (2013) – but using scipy.stats.kendalltau

C_nonchance()

Calculate the C for non-chance games (using the total KTD)

class common_interest.**CommonInterest_2_pops**(*game*)

Calculate quantities useful for the study of the degree of common interest between senders and receivers

K(*array*)

Calculate K as defined in Godfrey-Smith and Martinez (2013) – but using scipy.stats.kendalltau

sender_K()

Calculate K for the sender

receiver_K()

Calculate K for the receiver

C_chance()

Calculate C as defined in Godfrey-Smith and Martinez (2013) – but using scipy.stats.kendalltau

C_nonchance()

Calculate the C for non-chance games (using the total KTD)

common_interest.**C**(*vector1*, *vector2*)

Calculate C for two vectors

common_interest.**tau**(*vector1*, *vector2*)

Calculate the Kendall tau statistic among two vectors

common_interest.**intra_tau**(*unconds*, *array*)

Calculate the average (weighted by <unconds> of the pairwise Kendall's tau distance between rows (states) of <array>

common_interest.**total_tau**(*array1*, *array2*)

Calculate the KTD between the flattened <array1> and <array2>. Useful for NonChance games

common_interest.**tau_per_rows**(*unconds*, *array1*, *array2*)

Calculate the average (weighted by <unconds> of the Kendall's tau distance between the corresponding rows (states) of <array1> and <array2>

class common_interest.**Nash**(*game*)

Calculate Nash equilibria

is_Nash(*sender*, *receiver*)

Find out if sender and receiver are a Nash eqb

common_interest.**stability**(*array*)

Compute a coarse grained measure of the stability of the array

common_interest.**stable_vector**(*vector*)

Return true if the vector does not move

common_interest.**periodic_vector**(*vector*)

We take the FFT of a vector, and eliminate all components but the two main ones (i.e., the static and biggest sine amplitude) and compare the reconstructed wave with the original. Return true if close enough

3.6 Calculations relating to information theory info.py

Information-theoretic analyses

class info.**Information**(*game*, *sender*, *receiver*)

Calculate information-theoretic quantities between strats. It expects a game, as created by game.Chance or game.NonChance, a sender strategy, and a receiver strategy

mutual_info_states_acts()

Calculate the mutual info between states and acts

mutual_info_states_messages()

Calculate the mutual info between states and messages

mutual_info_messages_acts()

Calculate the mutual info between messages and acts

class info.**RDT**(*game*, *dist_tensor=None*, *epsilon=0.001*)

Calculate the rate-distortion function for a game and any number of distortion measures

dist_tensor_from_game()

Return normalize_distortion() for sender and receiver payoffs

blahut(*lambda_DUMMY*, *max_rounds=100*, *return_cond=False*)

Calculate the point in the R(D, D') surface with slopes given by lambda_DUMMY and mu. Follows Cover & Thomas 2006, p. 334

update_conditional(*lambda_DUMMY*, *output*)

Calculate a new conditional distribution from the <output> distribution and the <lambda_DUMMY> parameters. The conditional probability matrix is such that cond[i, j] corresponds to P(x^_j | x_i)

calc_distortion(*cond*, *matrix*)

Calculate the distortion for a given channel (individuated by the conditional matrix in <cond>), for a certain slice of self.dist_tensor

calc_rate(*cond*, *output*)

Calculate the rate for a channel (given by <cond>) and output distribution (given by <output>)

from_cond_to_RD(*cond*, *dist_measures*)

Take a channel matrix, cond, where cond[i, j] gives P(q^[j] | q[i]), and calculate rate and distortions for it. <dist_measures> is a list of integers stating which distortion measures we want.

class info.**OptimizeRate**(*game*, *dist_measures=None*, *dist_tensor=None*, *epsilon=0.0001*)

A class to calculate rate-distortion surface with a scipy optimizer

make_calc_RD()

Return a function that calculates an RD (hyper-)surface using the trust-constr scipy optimizer, for a given list of distortion objectives.

rate(*cond_flat*)

Calculate rate for make_calc_RD()

cond_init()

Return an initial conditional matrix

gen_lin_constraint(*distortions*)

Generate the LinearConstraint object

Parameters
distortions (*A list of distortion objectives*) –

lin_constraint()

Collate all constraints

dist_constraint()

Present the distortion constraint (which is linear) the way scipy.optimize expects it

prob_constraint()

Present the constraint that all rows in cond be probability vectors. We use a COO sparse matrix

class info.**OptimizeMessages**(*game*, *dist_measures=None*, *dist_tensor=None*, *epsilon=0.0001*)

A class to calculate number-of-messges/distortion curves with a scipy optimizer

make_calc_MD()

Return a function that calculates the minimum distortion attainable for a certain number of messages, using a trust-constr scipy optimizer. Right now it only works for one distortion measure. <distortion> is the distortion matrix in <dist_tensor> that we should care about.

distortion(*codec_flat*, *messages*, *matrix*)

Calculate the distortion for a given channel (individuated by the conditional matrix in <cond>), for a certain slice of self.dist_tensor

codec_init_random(*messages*)

Return an initial conditional matrix

codec_init(*messages*)

Return an initial conditional matrix

gen_lin_constraint(*messages*)

Generate the LinearConstraint object

Parameters
distortions (*A list of distortion objectives*) –

prob_constraint(*messages*)

Present the constraint that all rows in cond be probability vectors. We use a COO sparse matrix

class info.**OptimizeMessageEntropy**(*game*, *dist_measures=None*, *dist_tensor=None*, *messages=None*, *epsilon=0.0001*)

A class to calculate rate-distortion (where rate is actually the entropy of messages with a scipy optimizer).

make_calc_RD()

Return a function that calculates an RD (hyper-)surface using the trust-constr scipy optimizer, for a given list of distortion objectives.

minimize_distortion(*matrix*)

Return a function that finds an encoder-decoder pair, with the requisite dimension, that minimizes a single distortion objective, using a trust-constr scipy optimizer

message_entropy(*encode_decode*)

Calculate message entropy given an encoder-decoder pair, where the two matrices are flattened and then concatenated

enc_dec_init()

Return an initial conditional matrix

gen_nonlin_constraint(*distortions*)

Generate a list of NonLinearConstraint objects

Parameters
distortions (*A list of distortion objectives*) –

gen_lin_constraint()

Generate the LinearConstraint object

Parameters
distortions (*A list of probability objectives*) –

gen_dist_func(*matrix*)

Return the function that goes into the NonLinearConstraint objects

prob_constraint()

Present the constraint that all rows in encoder and decoder be probability vectors

class info.**Shea**(*game*)

Calculate functional content vectors as presented in Shea, Godfrey-Smith andand Cao 2017.

baseline_payoffs()

Give a vector with the payoffs for sender, and another for receiver, when the receiver does the best possible act for it in the absence of any communication. I will choose, for now, the receiver act that gives the best possible sender payoff (this is not decided by Shea et al.; see fn.14)

expected_for_act(*act*)

Calculate the expected payoff for sender and receiver of doing one act, in the absence of communication

normal_payoffs()

Calculate payoffs minus the baseline

calc_dmin()

Calculate dmin as defined in Shea et al. (2917, p. 24)

calc_summation(*norm_payoff*, *receiver_strat*)

Calculate the summation in the entries of the functional content vector

calc_entries(*sender_strat*, *receiver_strat*, *payoff_matrix*)

Calculate the entries of the functional vector, given one choice for the (baselined) payoff matrix

calc_entries_dmin(*sender_strat*, *receiver_strat*)

Calculate the entries of the functional vector, given one choice for the official dmin

calc_entries_sender(*sender_strat*, *receiver_strat*)

Calculate the entries of the functional vector, for the baselined sender

calc_entries_receiver(*sender_strat*, *receiver_strat*)

Calculate the entries of the functional vector, for the baselined receiver

calc_condition(*receiver_strat*, *payoff_matrix*, *baseline*)

Calculate the condition for nonzero vector entries

calc_condition_sender(*receiver_strat*)

Calculate condition() for the sender payoff matrix and baseline

calc_condition_receiver(*receiver_strat*)

Calculate condition() for the receiver payoff matrix and baseline

calc_condition_common(*receiver_strat*)

Calculate the condition for a nonzero functional vector entry in the definition in (op. cit., p. 24)

functional_content(*entries*, *condition*)

Put everything together in a functional vector per message

functional_content_sender(*sender_strat*, *receiver_strat*)

Calculate the functional content from the perspective of the sender

functional_content_receiver(*sender_strat*, *receiver_strat*)

Calculate the functional content from the perspective of the receiver

functional_content_dmin(*sender_strat*, *receiver_strat*)

Calculate the functional content from the perspective of dmin

info.**conditional_entropy**(*conds*, *unconds*)

Take a matrix of probabilities of the column random variable (r. v.) conditional on the row r.v.; and a vector of unconditional probabilities of the row r. v.. Calculate the conditional entropy of column r. v. on row r. v. That is: Input:

```
>>> [[P(B1|A1), ...., P(Bn|A1)],..., [P(B1|Am),...,P(Bn|Am)]]
>>> [P(A1), ..., P(Am)]
```

Output:

```
>>> H(B|A)
```

info.**mutual_info_from_joint**(*matrix*)

Take a matrix of joint probabilities and calculate the mutual information between the row and column random variables

info.**unconditional_probabilities**(*unconditional_input*, *strategy*)

Calculate the unconditional probability of messages for sender, or signals for receiver, given the unconditional probability of states (for sender) or of messages (for receiver)

info.**normalize_axis**(*array*, *axis*)

Normalize a matrix along <axis>, being sensible with all-zero rows

info.**from_joint_to_conditional**(*array*)

Normalize row-wise

info.**from_conditional_to_joint**(*unconds*, *conds*)

Take a matrix of conditional probabilities of the column random variable on the row r. v., and a vector of unconditional probabilities of the row r. v. and output a matrix of joint probabilities.

Input: >>> [[P(B1|A1), ..., P(Bn|A1)],..., [P(B1|Am),...,P(Bn|Am)]] >>> [P(A1), ..., P(Am)] Output: >>> [[P(B1,A1), ..., P(Bn,A1)],..., [P(B1,Am),...,P(Bn,Am)]]

info.**bayes_theorem**(*unconds*, *conds*)

Perform Bayes' theorem on a matrix of conditional probabilities

Parameters

- **unconds** (*a (n x 1) numpy array of unconditional probabilities [P(A1), ... , P(An)]*) –
- **conds** (*a (m x n) numpy array of conditional probabilities [[P(B1|A1), ... , P(Bm|A1)], ... , [P(B1|An), ..., P(Bm|An)]]*) –

Return type

A (n x m) numpy array of conditional probabilities [[P(A1|B1), … , P(An|B1)], … , [P(PA1|Bm), … , P(An|Bm)]]

info.**entropy**(*vector*)

Calculate the entropy of a vector

info.**escalar_product_map**(*matrix*, *vector*)

Take a matrix and a vector and return a matrix consisting of each element of the vector multiplied by the corresponding row of the matrix

info.**normalize_vector**(*vector*)

Normalize a vector, converting all-zero vectors to uniform ones

info.**normalize_distortion**(*matrix*)

Normalize linearly so that max corresponds to 0 distortion, and min to 1 distortion It must be a matrix of floats!

3.7 Exceptions exceptions.py

Some custom exceptions and errors

exception exceptions.**ChanceNodeError**

Error to raise when the user is attempting to do something with a chance node that doesn't exist

exception exceptions.**NoDataException**

Error to raise when the user tries to show a plot but the figure object doesn't have the required data. Also raised when the user tries to load data from disk that is not found.

exception exceptions.**InconsistentDataException**

Error to raise when the user provides data that is inconsistent e.g. payoff matrices that do not have a shape corresponding to the set of states or set of acts.

exception exceptions.**ModuleNotInstalledException**

Error to raise when a method requires a module that is not yet installed.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V