EGTtools Documentation

Release 0.1.6.dev1

Elias Fernández

Nov 05, 2021

CONTENTS

EGTtools provides a centralized repository with analytical and numerical methods to study/model game theoretical problems under the Evolutionary Game Theory (EGT) framework.

This library is composed of two parts:

- a set of analytical methods implemented in Python 3
- a set of computational methods implemented in C++ with (Python 3 bindings)

The second typed is used in cases where the state space is too big to solve analytically, and thus require estimating the model parameters through monte-carlo simulations. The C++ implementation provides optimized computational methods that can run in parallel in a reasonable time, while Python bindings make the methods easily accecible to a larger range of researchers.

CHAPTER

ONE

EXAMPLES

The following examples give an idea of how to use EGTtools to model social dynamics:

1.1 Example of use

[1]:
```
import numpy as np
# Plotting libraries
import matplotlib.pylab as plt
# Magic function to make matplotlib inline; other style specs must come AFTER
%matplotlib inline
# This enables high resolution PNGs.
%config InlineBackend.figure_formats = {'png', 'svg'}
```

1.1.1 Evolutionary Dynamics of the Haw-Dove Game in Infinite populations

Replicator equation

The replicator equation represents the dynamics of competing individuals in a population. It normally found in the following form:

$\dot{x}_i = x_i[f(x_i) - \sum_{i=1}^{n} x_i f(x_i)]$,

where x_i represents the frequency of strategy i in the population, and $f(x_i)$ is the fitness of strategy i. This differencial equation, gives the gradient of selection, i.e., the strength with which the frequency of a certain strategy will increase or decrease. It may also be expressed in a more convenient matrix form:

$G(x_i) = \dot{x}_i = x_i[(Ax)_i - x^T Ax]$

Where the matrix A is a payoff matrix with element A_{ij} representing the fitness of strategy i over strategy j.

[2]:
```
from egttools.analytical import replicator_equation
from egttools.utils import find_saddle_type_and_gradient_direction
from egttools.plotting import plot_gradient
```

[3]:
```
nb_points = 101
strategy_i = np.linspace(0, 1, num=nb_points, dtype=np.float64)
strategy_j = 1 - strategy_i
states = np.array((strategy_i,strategy_j)).T

# Payoff matrix
```

(continues on next page)

(continued from previous page)

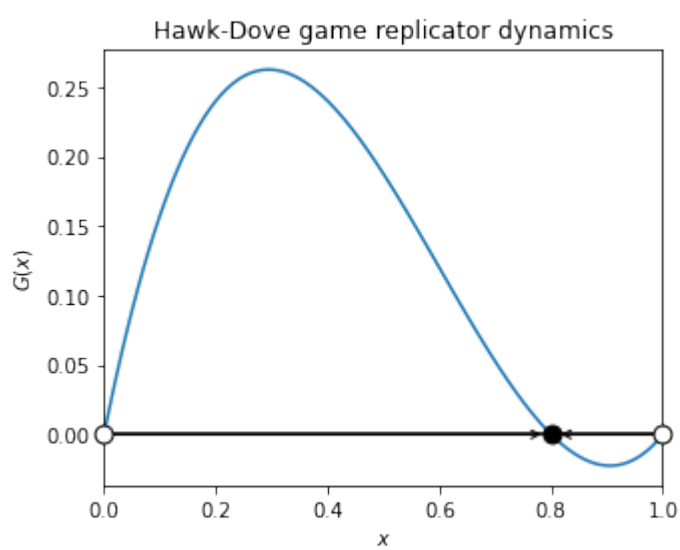
```
V = 2; D = 3; T = 1
A = np.array([
        [ (V-D)/2, V],
        [ 0      , (V/2) - T],
    ])
```

[4]:
```
# Calculate gradient
G = np.array([replicator_equation(states[i], A)[0] for i in range(len(states))])
```

[5]:
```
# Find saddle points (where the gradient is 0)
epsilon = 1e-7
saddle_points_idx = np.where((G <= epsilon) & (G >= -epsilon))[0]
saddle_points = saddle_points_idx / (nb_points - 1)

# Now let's find which saddle points are absorbing/stable and which aren't
# we also annotate the gradient's direction among saddle poinst
saddle_type, gradient_direction = find_saddle_type_and_gradient_direction(G, saddle_
↪points_idx)
```

[6]:
```
ax = plot_gradient(strategy_i,
                   G,
                   saddle_points,
                   saddle_type,
                   gradient_direction,
                   'Hawk-Dove game replicator dynamics',
                   xlabel='$x$')
plt.show()
```



1.1.2 Evolutionary Dynamics of the Hawk-Dove Game in Finite populations

Now we are going to study the effect of having Finite populations. In general, finite populations introduce stochastic effects in the dynamics, also known as random drift $1/Z$, where Z is the size of the population. We can represent these dynamics, by adapting the replicator equation, which considers that individuals are sampled from an infinite population, and therefore selecting a member of strategy j does not reduce its frequency in the population. When the population is finite, we make no longer assume a sampling with replacement, i.e., when an individual of strategy j is sampled, the fraction of members of that strategy is reduced, instead we must sample without replacement.

Here the fitness of an strategy i against strategy j directly depends on the size of the population:

$f_i(x_i, Z) = \frac{x_i - 1}{Z-1} * A_{ii} + \frac{Z - x_i}{Z-1} * A_{ij}$

For the selection dynamics, we use a Moran process (or birth-death process) with pair-wise comparison: at each step, 2 individuals, a and b, are randomly sampled (without replacement) from the population, and their payoff is compared. The fermi equation gives the probability that individual a (selected for death) will copy the strategy of individual b (selected for birth):

$p = [1 + e^{\beta(f_a - f_b)}]^{-1}$

a will imitate b with probability p, in any other case, the population state will not change. β indicates the selection strength and on the limit $\beta \to 0$ all strategies are immitated with equal probability.

[7]:
```
from egttools.analytical import StochDynamics
```

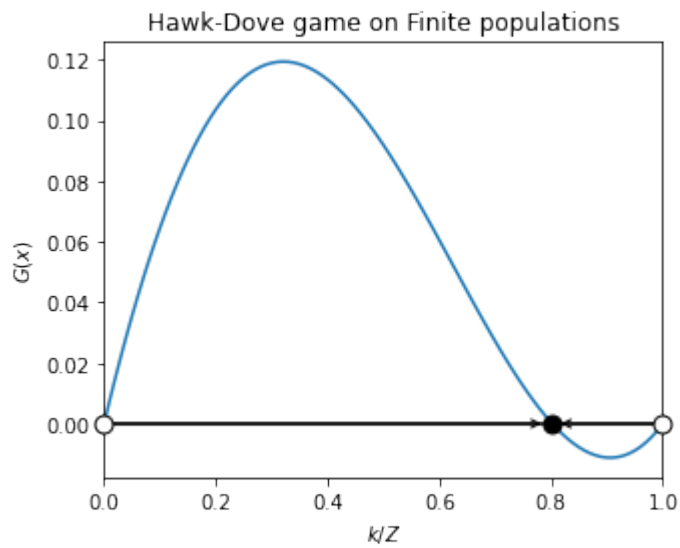
[8]:
```
# Parameters and evolver
nb_strategies = 2; Z = 100; N = 2;
beta = 1
pop_states = np.arange(0, Z + 1, 1)
strategy_i = np.linspace(0, 1, num=Z + 1, dtype=np.float64)
evolver = StochDynamics(nb_strategies, A, Z)
```

[9]:
```
gradients = np.array([evolver.gradient_selection(x, 0, 1, beta)
                      for x in pop_states])
```

[10]:
```
# Find saddle points (where the gradient is 0) - Define epsilon correctly, or you will␣
↪get wrong matches
epsilon = 1e-3
saddle_points_idx = np.where((gradients <= epsilon) & (gradients >= -epsilon))[0]
saddle_points = saddle_points_idx / Z

# Now let's find which saddle points are absorbing/stable and which aren't
# we also annotate the gradient's direction among saddle poinst
saddle_type, gradient_direction = find_saddle_type_and_gradient_direction(gradients,
                                                                          saddle_points_
↪idx)
```

[18]:
```
ax = plot_gradient(strategy_i,
                   gradients,
                   saddle_points,
                   saddle_type,
                   gradient_direction,
                   'Hawk-Dove game on Finite populations',
                   xlabel='$k/Z$')
plt.show()
```



[12]:
```
evolver.mu = 0
stationary_SML = evolver.calculate_stationary_distribution(beta)
print("time spent as Hawk: {} & time spent as Dove: {}".format(*stationary_SML))
```

```
time spent as Hawk: 1.0 & time spent as Dove: 0.0
```

If we introduce mutations, i.e., there is a probability that individuals make an error, and adopt a different strategy instead of imitating the best, the dynamics may change. In this case, at each time step, players are selected for death/birth and their fitness is compared.

However, now, with probability μ agent a will adopt a random strategy from the strategy space, and with probability $1 - \mu$ it will immitate b with probability p. Therefore the probability of immitating b is $p_{eff} = (1 - \mu) * p$. On the limit $\mu \to 1$, all strategies are taken with equal probability. When $\mu \to 0$ we go back to the previous case, also known as small mutation limit (SML).

When mutation is small, we may assume that only the the states in which all population adopts a single strategy, also known as monomorphic states, are absorbing and stable. This is because, since there are no mutations, once the population reaches a monomorphic state, it will never leave. Such a simplification, allows us to reduce the number of states of the system, taking the Hawk-Dove game as an example, from Z + 1, to only 2.

Moreover, mixed equilibria are no longer stable in the SML. This occurs, since random drift, even if it takes an infinite ammount of time (please note that we are not looking at fixation times), will drive the population to one of the monomorphic states. For this reason, the SML assumption is only reasonable when we know that there are no mixed stable attractors in the studied system, which is not the case in the Hawk-Dove game. This explains why the results of the stationary distribution differ from the previously calculated Nash equilibria.

Now we are going to calculate the stationary distribution again, but taking mutations into account.

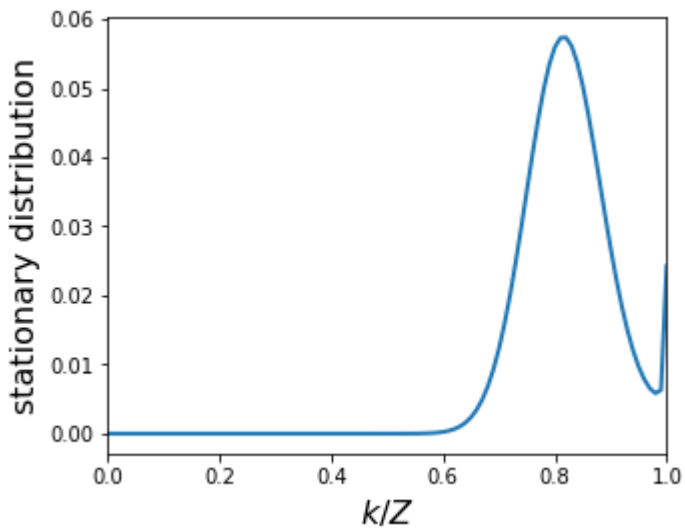
[13]:
```
evolver.mu = 1e-3
stationary_with_mu = evolver.calculate_stationary_distribution(beta)
```

[15]:
```
fig, ax = plt.subplots(figsize=(5, 4))
fig.patch.set_facecolor('white')
lines = ax.plot(np.arange(0, Z+1)/Z, stationary_with_mu)
plt.setp(lines, linewidth=2.0)
ax.set_ylabel('stationary distribution',size=16)
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel('$k/Z$',size=16)
ax.set_xlim(0, 1)
plt.show()
```



[ ]:

```

```

[1]:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import os

# Ony necessary in MacOSX and Anaconda due to known error with duplicated symbol
# when using OpenMP
# This line can also be avoided installing anaconda's nomkl (conda install nomkl)
os.environ['KMP_DUPLICATE_LIB_OK']='True'
```

[2]:

```
import egttools as egt
```

[3]:

```
egt.numerical.Random.init()
seed = egt.numerical.Random.seed_
```

[4]:

```
# Payoff matrix
V = 2; D = 3; T = 1
A = np.array([
        [ (V-D)/2, V],
        [ 0      , (V/2) - T],
    ])
```

[71]:

```
A
```

[71]:

```
array([[-0.5,  2. ],
       [ 0. ,  0. ]])
```

[5]:
```
game = egt.numerical.games.NormalFormGame(1, A)
```

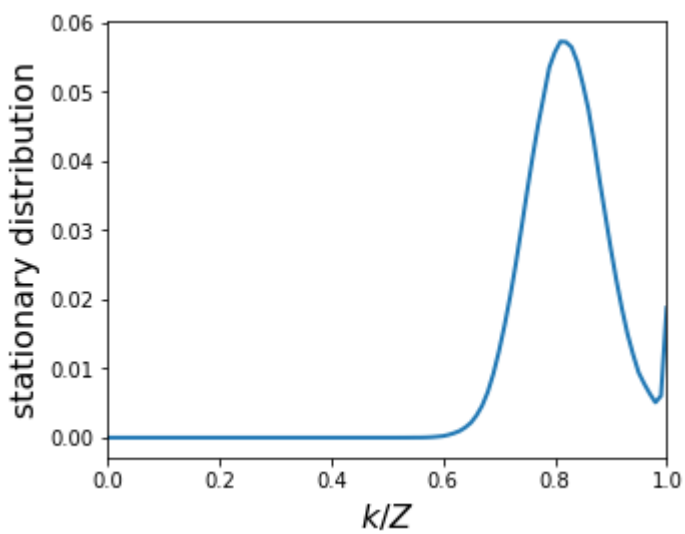
[6]:
```
Z = 100
x = np.arange(0, Z+1)/Z
```

[7]:
```
evolver = egt.numerical.PairwiseMoran(Z, game, 1000000)
```

[8]:
```
Z = 100
x = np.arange(0, Z+1)/Z
evolver.pop_size = Z
```

[9]:
```
dist = evolver.stationary_distribution(10, int(1e6), int(1e3), 1, 1e-3)
```

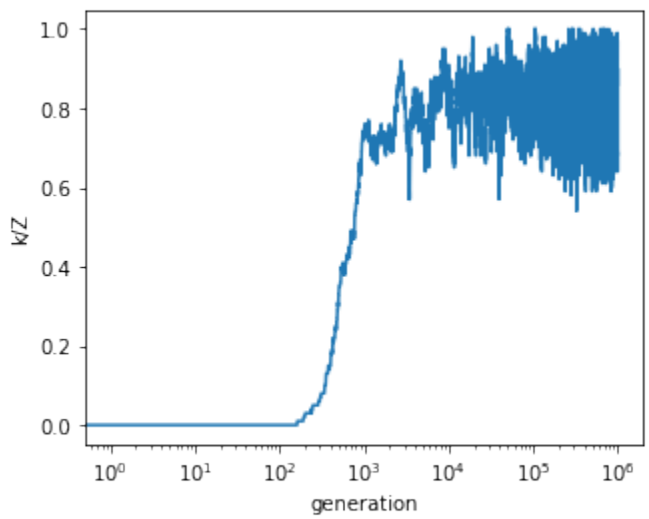
[10]:
```
# We need to reverse, since in this case we are starting from the case
# where the number of Haws is 100%, because of how we map states
fig, ax = plt.subplots(figsize=(5, 4))
fig.patch.set_facecolor('white')
lines = ax.plot(x, list(reversed(dist)))
plt.setp(lines, linewidth=2.0)
ax.set_ylabel('stationary distribution',size=16)
ax.set_xlabel('$k/Z$',size=16)
ax.set_xlim(0, 1)
plt.show()
```



1.2 We can also plot a single run

[11]:
```
output = evolver.run(int(1e6), 1, 1e-3, [0, Z])
```

[12]:
```
fig, ax = plt.subplots(figsize=(5, 4))
ax.plot(output[:, 0]/Z)
ax.set_ylabel('k/Z')
ax.set_xlabel('generation')
ax.set_xscale('log')
plt.show()
```



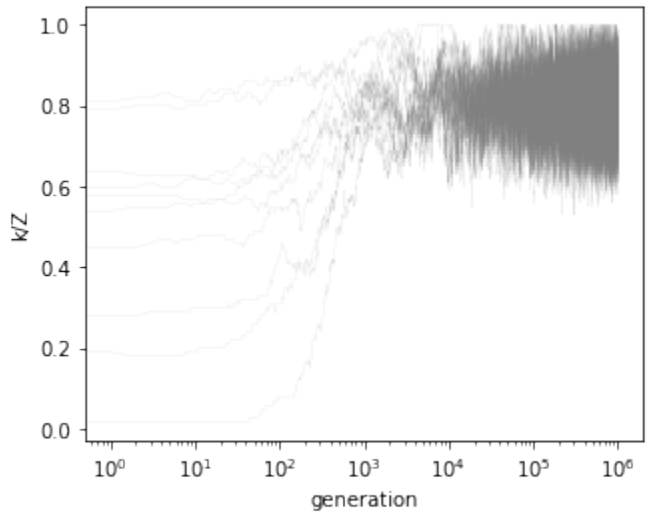
1.3 And can visualize what happens if we start several runs from random points in the simplex

[13]:
```
init_states = np.random.randint(0, Z+1, size=10, dtype=np.uint64)
```

[14]:
```
output = []
for i in range(10):
    output.append(evolver.run(int(1e6), 1, 1e-3,
                              [init_states[i], Z - init_states[i]]))
```

[70]:
```
# Plot each year's time series in its own facet
fig, ax = plt.subplots(figsize=(5, 4))

for run in output:
    ax.plot(run[:, 0]/Z, color='gray', linewidth=.1, alpha=0.6)
ax.set_ylabel('k/Z')
ax.set_xlabel('generation')
ax.set_xscale('log')
```



1.4 And we can also compare how far our approximations are from the analytical results

[16]:
```
# We do this for different betas
betas = np.logspace(-4, 1, 50)
```

[17]:
```
stationary_points = []
for i in range(len(betas)):
    stationary_points.append(evolver.stationary_distribution(30, int(1e6), int(1e3),
                                                             betas[i], 1e-3))
stationary_points = np.asarray(stationary_points)
```

[18]:
```
# Now we estimate the probability of Cooperation for each possible state
state_frequencies = np.arange(0, Z+1) / Z
coop_level = np.dot(state_frequencies, stationary_points.T)
```

[19]:
```
# Finnally we do the same, but for the analytical results
from egttools.analytical import StochDynamics
```

[20]:
```
analytical_evolver = egt.analytical.StochDynamics(2, A, Z, mu=1e-3)
```

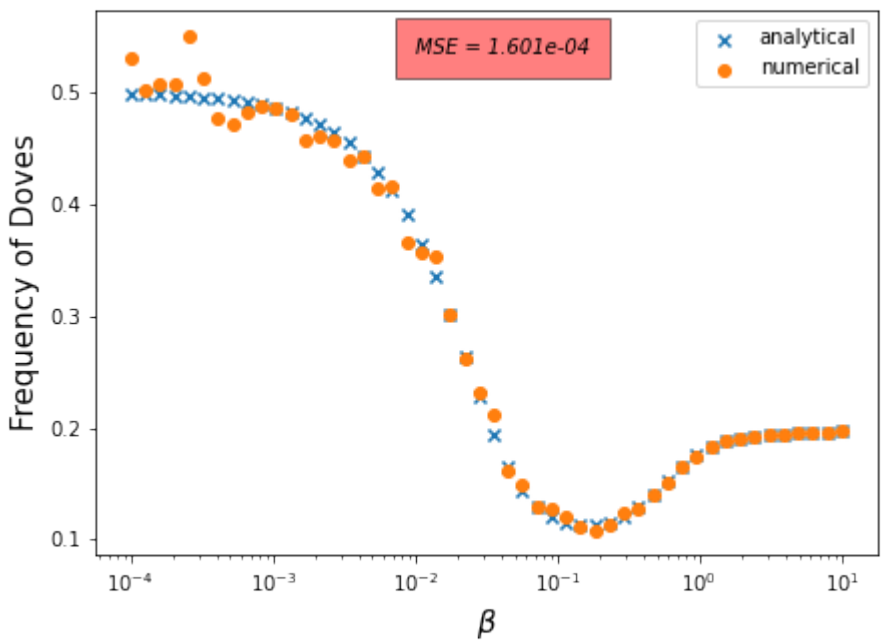
[21]:
```
stationary_points_analytical = []
for i in range(len(betas)):
    stationary_points_analytical.append(analytical_evolver.calculate_stationary_
↪distribution(betas[i]))
stationary_points_analytical = np.asarray(stationary_points_analytical)
```

[22]:
```
# Now we estimate the probability of Cooperation for each possible state
coop_level_analytical = np.dot(1 - state_frequencies, stationary_points_analytical.T)
```

[25]:
```
from sklearn.metrics import mean_squared_error
```

[26]:
```
mse = mean_squared_error(coop_level_analytical, coop_level)
```

[74]:
```
# Finally, we plot and compare visually (and check how much error we get)
fig, ax = plt.subplots(figsize=(7, 5))
# ax.scatter(betas, coop_level, label="simulation")
ax.scatter(betas, coop_level_analytical, marker='x', label="analytical")
ax.scatter(betas, coop_level, marker='o', label="numerical")
ax.text(0.01, 0.535, 'MSE = {0:.3e}'.format(mse), style='italic',
        bbox={'facecolor': 'red', 'alpha': 0.5, 'pad': 10})
ax.legend()
ax.set_xlabel(r'$\beta$', fontsize=15)
ax.set_ylabel('Frequency of Doves', fontsize=15)
ax.set_xscale('log')
plt.show()
```



[ ]:

1.5 Projects using EGTtools

The EvoCRD provide an example of use of EGTtools do model dynamics in the Collective Risk Dilemma.

CHAPTER

TWO

EGTTOOLS

The `egttools` package implements methods to study evolutionary dynamics.

Functions

`calculate_nb_states`	Calculates the number of states (combinations) of the members of a group in a subgroup.
`calculate_state`	Overloaded function.
`calculate_strategies_distribution`	Calculates the average frequency of each strategy available in the population given the stationary distribution.
`sample_simplex`	Transforms a state index into a vector.

2.1 egttools.calculate_nb_states

calculate_nb_states(*group_size: int*, *nb_strategies: int*) → int

Calculates the number of states (combinations) of the members of a group in a subgroup. It can be used to calculate the maximum number of states in a discrete simplex.

The implementation of this method follows the stars and bars algorithm (see Wikipedia).

Parameters

- **group_size** (*int*) – Size of the group (maximum number of players/elements that can adopt each possible strategy).
- **nb_strategies** (*int*) – number of strategies that can be assigned to players.

Returns Number of states (possible combinations of strategies and players).

Return type int

See also:

`egttools.numerical.calculate_state`, `egttools.numerical.sample_simplex`

2.2 egttools.calculate_state

calculate_state(**args*, ***kwargs*)

Overloaded function.

1. calculate_state(group_size: int, group_composition: List[int]) -> int

   This function converts a vector containing counts into an index.

   This method was copied from @Svalorzen.

   group_size [int] Maximum bin size (it can also be the population size).

   group_composition [List[int]] The vector to convert from simplex coordinates to index.

   int The unique index in [0, egttools.calculate_nb_states(group_size, len(group_composition)) representing the n-dimensional simplex.

   egttools.sample_simplex, egttools.calculate_nb_states

2. calculate_state(group_size: int, group_composition: numpy.ndarray[numpy.uint64[m, 1]]) -> int

   This function converts a vector containing counts into an index.

   This method was copied from @Svalorzen.

   group_size [int] Maximum bin size (it can also be the population size).

   group_composition [numpy.ndarray[numpy.int64[m, 1]]] The vector to convert from simplex coordinates to index.

   int The unique index in [0, egttools.calculate_nb_states(group_size, len(group_composition)) representing the n-dimensional simplex.

   egttools.sample_simplex, egttools.calculate_nb_states

2.3 egttools.calculate_strategies_distribution

calculate_strategies_distribution(*pop_size: int*, *nb_strategies: int*, *stationary_distribution: scipy.sparse.csr_matrix[numpy.float64]*) → numpy.ndarray[numpy.float64[m, 1]]

Calculates the average frequency of each strategy available in the population given the stationary distribution.

Parameters

- **pop_size** (*int*) – Size of the population.
- **nb_strategies** (*int*) – Number of strategies that can be assigned to players.
- **stationary_distribution** (*scipy.sparse.csr_matrix*) – A sparse matrix which contains the stationary distribution (the frequency with which the evolutionary system visits each stationary state).

Returns Average frequency of each strategy in the stationary evolutionary system.

Return type numpy.ndarray[numpy.float64[m, 1]]

See also:

egttools.numerical.calculate_state, *egttools.numerical.calculate_nb_states*, *egttools.numerical.sample_simplex*, *egttools.numerical.PairwiseMoran.stationary_distribution_sparse*

2.4 egttools.sample_simplex

sample_simplex(*index: int*, *pop_size: int*, *nb_strategies: int*) → numpy.ndarray[numpy.uint64[m, 1]]

Transforms a state index into a vector.

Parameters

- **index** (*int*) – State index.
- **pop_size** (*int*) – Size of the population.
- **nb_strategies** (*int*) – Number of strategies.

Returns Vector with the sampled state.

Return type numpy.ndarray[numpy.int64[m, 1]]

See also:

egttools.numerical.calculate_state, *egttools.numerical.calculate_nb_states*

Classes

Random	Random seed generator.

2.5 egttools.Random

class Random

Bases: pybind11_builtins.pybind11_object

Random seed generator.

Methods

generate	Generates a random seed.
init	Overloaded function.
seed	This static methods changes the seed of *egttools.Random*.

__init__(**args*, ***kwargs*)

__new__(***kwargs*)

static generate() → int

Generates a random seed.

The generated seed can be used to seed other pseudo-random generators, so that the initial state of the

simulation can always be tracked and the simulation can be reproduced. This is very important both for debugging purposes as well as for scientific research. However, this approach should NOT be used in any cryptographic applications, it is NOT safe.

Returns A random seed which can be used to seed new random generators.

Return type int

static init(**args*, ***kwargs*)

Overloaded function.

1. init() -> egttools.numerical.Random

   This static method initializes the random seed generator from random_device and returns an instance of egttools.Random which is used to seed the random generators used across egttools.

   egttools.Random An instance of the random seed generator.

2. init(seed: int) -> egttools.numerical.Random

   This static method initializes the random seed generator from seed and returns an instance of `egttools.Random` which is used to seed the random generators used across `egttools`.

   seed [int] Integer value used to seed the random generator.

   egttools.Random An instance of the random seed generator.

static seed(*seed: int*) → None

This static methods changes the seed of `egttools.Random`.

Parameters **int** – The new seed for the `egttools.Random` module which is used to seed every other pseudo-random generation in the `egttools` package.

`egttools.analytical`	API reference documentation for `sed_analytical` submodule.
`egttools.behaviors`	API reference documentation for `behaviors` submodule.
`egttools.games`	API reference documentation for the `games` submodule.
`egttools.numerical`	The `numerical` module contains optimized functions and classes to simulate evolutionary dynamics in large populations.
`egttools.plotting`	API reference documentation for the `plotting` submodule.
`egttools.utils`	This python module contains some utility functions to find saddle points and plot gradients in 2 player, 2 strategy games.

2.6 egttools.analytical

API reference documentation for `sed_analytical` submodule.

Functions

`replicator_equation`	Produces the discrete time derivative of the replicator dynamics

2.6.1 egttools.analytical.replicator_equation

replicator_equation(*x*, *payoffs*)

Produces the discrete time derivative of the replicator dynamics

Parameters

- **x** (*numpy.ndarray[numpy.float64[m,1]]*) – array containing the frequency of each strategy in the population.
- **payoffs** (*numpy.ndarray[numpy.float64[m,m]]*) – payoff matrix

Returns time derivative of x

Return type numpy.ndarray

See also:

egttools.analytical.StochDynamics, *egttools.numerical.PairwiseMoran*

Classes

`StochDynamics`	A class containing methods to calculate the stochastic evolutionary dynamics of a population.

2.6.2 egttools.analytical.StochDynamics

class StochDynamics(*nb_strategies*, *payoffs*, *pop_size*, *group_size=2*, *mu=0*)

Bases: `object`

A class containing methods to calculate the stochastic evolutionary dynamics of a population.

Defines a class that contains methods to compute the stationary distribution for the limit of small mutation (only the monomorphic states) and the full transition matrix.

Parameters

- **nb_strategies** (*int*) – number of strategies in the population
- **pop_size** (*int*) – population size
- **payoffs** (*numpy.ndarray[numpy.float64[m,m]]*) – payoff matrix indicating the payoff of each strategy (rows) against each other (columns)
- **mu** (*float*) – mutation probability

See also:

egttools.numerical.PairwiseMoran, *egttools.analytical.replicator_equation*

Methods

calculate_full_transition_matrix	Returns the full transition matrix in sparse representation.
calculate_stationary_distribution	Calculates the stationary distribution of the monomorphic states is mu = 0 (SML).
fermi	The fermi function determines the probability that the first type imitates the second.
fitness_group	In a population of x i-strategists and (Z-x) j strategists, where players interact in group of 'group_size' participants this function returns the average payoff of strategies i and j.
fitness_pair	Calculates the fitness of strategy i versus strategy j, in a population of x i-strategists and (Z-x) j strategists, considering a 2-player game.
fixation_probability	Function for calculating the fixation_probability probability of the invader in a population of residents.
full_fitness_difference_group	Calculate the fitness difference between strategies :param i and :param j assuming that player interact in groups of size N > 2 (n-player games).
full_fitness_difference_pairwise	Calculates the fitness of strategy i in a population with state :param population_state, assuming pairwise interactions (2-player game).
full_gradient_selection	Calculates the gradient of selection for an invading strategy, given a population state.
full_prob_increase_decrease_with_mutation	Calculates the probabilities of increasing/decreasing the frequency of a strategy in each possible population state.
gradient_selection	Calculates the gradient of selection given an invader and a resident strategy.
prob_increase_decrease	This function calculates for a given number of invaders the probability that the number increases or decreases with one.
prob_increase_decrease_with_mutation	This function calculates for a given number of invaders the probability that the number increases or decreases with taking into account a mutation rate.
transition_and_fixation_matrix	Calculates the transition matrix (only for the monomorphic states) and the fixation_probability probabilities.

__init__(*nb_strategies*, *payoffs*, *pop_size*, *group_size=2*, *mu=0*)

calculate_full_transition_matrix(*beta*, **args*)

Returns the full transition matrix in sparse representation.

Parameters

- **beta** (*float*) – Intensity of selection.
- **args** (*Optional[list]*) – Other arguments. Can be used to pass extra arguments to

functions contained in the payoff matrix.

Returns The full transition matrix between the two strategies in sparse format.

Return type scipy.sparse.lil_matrix

`calculate_stationary_distribution`(*beta*, **args*)

Calculates the stationary distribution of the monomorphic states is mu = 0 (SML). Otherwise, it calculates the stationary distribution including all possible population states.

Parameters

- **beta** (*float*) – intensity of selection.
- **args** (*Optional[list]*) – extra arguments for calculating payoffs.

Returns A vector containing the stationary distribution

Return type numpy.ndarray

`static fermi`(*beta*, *fitness_diff*)

The fermi function determines the probability that the first type imitates the second.

Parameters

- **beta** (*float*) – intensity of selection
- **fitness_diff** (*float*) – Difference in fitneess between the strategies (f_a - f_b).

Returns the probability of imitation

Return type numpy.typing.ArrayLike

`fitness_group`(*x*, *i*, *j*, **args*)

In a population of x i-strategists and (Z-x) j strategists, where players interact in group of 'group_size' participants this function returns the average payoff of strategies i and j.

Parameters

- **x** (*int*) – number of individuals adopting strategy i in the population
- **i** (*int*) – index of strategy i
- **j** (*int*) – index of strategy j
- **args** (*Optional[list]*) – Other Parameters. This can be used to pass extra parameters to functions stored in the payoff matrix

Return type `float`

Returns

- *float*
- *Returns the difference in fitness between strategy i and j*

`fitness_pair`(*x*, *i*, *j*, **args*)

Calculates the fitness of strategy i versus strategy j, in a population of x i-strategists and (Z-x) j strategists, considering a 2-player game.

Parameters

- **x** (*int*) – number of i-strategists in the population
- **i** (*int*) – index of strategy i
- **j** (*int*) – index of strategy j
- **args** (*Optional[list]*) –

Return type `float`

Returns

- *float*
- *the fitness difference among the strategies*

`fixation_probability`(*invader*, *resident*, *beta*, **args*)

Function for calculating the fixation_probability probability of the invader in a population of residents.

Parameters

- **invader** (*int*) – index of the invading strategy
- **resident** (*int*) – index of the resident strategy
- **beta** (*float*) – intensity of selection
- **args** (*Optional[list]*) – Other arguments. Can be used to pass extra arguments to functions contained in the payoff matrix.

Returns The fixation_probability probability.

Return type float

See also:

egttools.numerical.PairwiseMoran

`full_fitness_difference_group`(*i*, *j*, *population_state*)

Calculate the fitness difference between strategies :param i and :param j assuming that player interact in groups of size N > 2 (n-player games).

Parameters

- **i** (*int*) – index of the strategy that will reproduce
- **j** (*int*) – index of the strategy that will die
- **population_state** (*numpy.ndarray[numpy.int64[m,1]]*) – vector containing the counts of each strategy in the population

Return type `float`

Returns

- *float*
- *The fitness difference between strategies i and j*

`full_fitness_difference_pairwise`(*i*, *j*, *population_state*)

Calculates the fitness of strategy i in a population with state :param population_state, assuming pairwise interactions (2-player game).

Parameters

- **i** (*int*) – index of the strategy that will reproduce
- **j** (*int*) – index of the strategy that will die
- **population_state** (*numpy.ndarray[numpy.int64[m,1]]*) – vector containing the counts of each strategy in the population

Return type `float`

Returns

- *float*

- *The fitness difference between the two strategies for the given population state*

full_gradient_selection(*population_state*, *beta*)

Calculates the gradient of selection for an invading strategy, given a population state.

Parameters

- **population_state** (*numpy.ndarray[np.int64[m,1]]*) – structure of unsigned integers containing the counts of each strategy in the population
- **beta** (*float*) – intensity of selection

Returns Matrix indicating the likelihood of change in the population given an starting point.

Return type numpy.ndarray[numpy.float64[m,m]]

full_prob_increase_decrease_with_mutation(*population_state*, *beta*)

Calculates the probabilities of increasing/decreasing the frequency of a strategy in each possible population state.

Parameters

- **population_state** (*numpy.ndarray[numpy.int64[m,1]]*) – structure of unsigned integers containing the counts of each strategy in the population
- **beta** (*float*) – intensity of selection

Returns

Returns an ndarray matrix with the probabilities of increasing/decreasing one individual adopting a given strategy in the current population state. All possible new state transition probabilities are returned.

e.g., given the population state (34, 33, 33) which represents a population of size 100 with 3 strategies - obviously the population state may be represented with a 2D tuple (34, 33) - then there are 6 possible changes to the population: (35, 32, 33), (35, 33, 32), (33, 34, 33), (33, 33, 34), (34, 34, 32), (34, 32, 34).

Return type numpy.ndarray[np.float64[m,m]]

gradient_selection(*k*, *invader*, *resident*, *beta*, **args*)

Calculates the gradient of selection given an invader and a resident strategy.

Parameters

- **k** (*int*) – number of invaders in the population
- **invader** (*int*) – index of the invading strategy
- **resident** (*int*) – index of the resident strategy
- **beta** (*float*) – intensity of selection
- **args** (*Optional[list[*) – other arguments. Can be used to pass extra arguments to functions contained in the payoff matrix.

Returns The gradient of selection.

Return type float

prob_increase_decrease(*k*, *invader*, *resident*, *beta*, **args*)

This function calculates for a given number of invaders the probability that the number increases or decreases with one.

Parameters

- **k** (*int*) – number of invaders in the population

- **invader** (*int*) – index of the invading strategy
- **resident** (*int*) – index of the resitent strategy
- **beta** (*float*) – intensity of selection
- **args** (*Optional[list]*) – other arguments. Can be used to pass extra arguments to functions contained in the payoff matrix.

Returns tuple(probability of increasing the number of invaders, probability of decreasing)

Return type Tuple[numpy.typing.ArrayLike, numpy.typing.ArrayLike]

prob_increase_decrease_with_mutation(*k*, *invader*, *resident*, *beta*, **args*)

This function calculates for a given number of invaders the probability that the number increases or decreases with taking into account a mutation rate.

Parameters

- **k** (*int*) – number of invaders in the population
- **invader** (*int*) – index of the invading strategy
- **resident** (*int*) – index of the resitent strategy
- **beta** (*float*) – intensity of selection
- **args** (*Optional[list]*) – other arguments. Can be used to pass extra arguments to functions contained in the payoff matrix.

Returns tuple(probability of increasing the number of invaders, probability of decreasing)

Return type Tuple[float, float]

transition_and_fixation_matrix(*beta*, **args*)

Calculates the transition matrix (only for the monomorphic states) and the fixation_probability probabilities.

This method calculates the transitions between monomorphic states. Thus, it assumes that we are in the small mutation limit (SML) of the moran process. Only use this method if this assumption is reasonable.

Parameters

- **beta** (*float*) – Intensity of selection.
- **args** (*Optional[list]*) – Other arguments. Can be used to pass extra arguments to functions contained in the payoff matrix.

Returns This method returns a tuple with the transition matrix as first element, and the matrix of fixation probabilities as second element.

Return type Tuple[numpy.ndarray[numpy.float64[m,m]], numpy.ndarray[numpy.float64[m,m]]]

egttools.analytical.sed_analytical	This python module contains the necessary functions to calculate analytically the evolutionary dynamics in Infinite and Finite populations on 2-player games.

2.6.3 egttools.analytical.sed_analytical

This python module contains the necessary functions to calculate analytically the evolutionary dynamics in Infinite and Finite populations on 2-player games.

Functions

calculate_nb_states	Calculates the number of states (combinations) of the members of a group in a subgroup.
replicator_equation	Produces the discrete time derivative of the replicator dynamics
sample_simplex	Transforms a state index into a vector.

egttools.analytical.sed_analytical.calculate_nb_states

calculate_nb_states(*group_size: int*, *nb_strategies: int*) → int

Calculates the number of states (combinations) of the members of a group in a subgroup. It can be used to calculate the maximum number of states in a discrete simplex.

The implementation of this method follows the stars and bars algorithm (see Wikipedia).

Parameters

- **group_size** (*int*) – Size of the group (maximum number of players/elements that can adopt each possible strategy).
- **nb_strategies** (*int*) – number of strategies that can be assigned to players.

Returns Number of states (possible combinations of strategies and players).

Return type int

See also:

egttools.numerical.calculate_state, *egttools.numerical.sample_simplex*

egttools.analytical.sed_analytical.replicator_equation

replicator_equation(*x*, *payoffs*)

Produces the discrete time derivative of the replicator dynamics

Parameters

- **x** (*numpy.ndarray[numpy.float64[m,1]]*) – array containing the frequency of each strategy in the population.
- **payoffs** (*numpy.ndarray[numpy.float64[m,m]]*) – payoff matrix

Returns time derivative of x

Return type numpy.ndarray

See also:

egttools.analytical.StochDynamics, *egttools.numerical.PairwiseMoran*

egttools.analytical.sed_analytical.sample_simplex

sample_simplex(*index: int*, *pop_size: int*, *nb_strategies: int*) → numpy.ndarray[numpy.uint64[m, 1]]

Transforms a state index into a vector.

Parameters

- **index** (*int*) – State index.
- **pop_size** (*int*) – Size of the population.
- **nb_strategies** (*int*) – Number of strategies.

Returns Vector with the sampled state.

Return type numpy.ndarray[numpy.int64[m, 1]]

See also:

egttools.numerical.calculate_state, *egttools.numerical.calculate_nb_states*

Classes

StochDynamics	A class containing methods to calculate the stochastic evolutionary dynamics of a population.
lil_matrix	Row-based list of lists sparse matrix

egttools.analytical.sed_analytical.StochDynamics

class StochDynamics(*nb_strategies*, *payoffs*, *pop_size*, *group_size=2*, *mu=0*)

Bases: object

A class containing methods to calculate the stochastic evolutionary dynamics of a population.

Defines a class that contains methods to compute the stationary distribution for the limit of small mutation (only the monomorphic states) and the full transition matrix.

Parameters

- **nb_strategies** (*int*) – number of strategies in the population
- **pop_size** (*int*) – population size
- **payoffs** (*numpy.ndarray[numpy.float64[m,m]]*) – payoff matrix indicating the payoff of each strategy (rows) against each other (columns)
- **mu** (*float*) – mutation probability

See also:

egttools.numerical.PairwiseMoran, *egttools.analytical.replicator_equation*

Methods

calculate_full_transition_matrix	Returns the full transition matrix in sparse representation.
calculate_stationary_distribution	Calculates the stationary distribution of the monomorphic states is mu = 0 (SML).
fermi	The fermi function determines the probability that the first type imitates the second.
fitness_group	In a population of x i-strategists and (Z-x) j strategists, where players interact in group of 'group_size' participants this function returns the average payoff of strategies i and j.
fitness_pair	Calculates the fitness of strategy i versus strategy j, in a population of x i-strategists and (Z-x) j strategists, considering a 2-player game.
fixation_probability	Function for calculating the fixation_probability probability of the invader in a population of residents.
full_fitness_difference_group	Calculate the fitness difference between strategies :param i and :param j assuming that player interact in groups of size N > 2 (n-player games).
full_fitness_difference_pairwise	Calculates the fitness of strategy i in a population with state :param population_state, assuming pairwise interactions (2-player game).
full_gradient_selection	Calculates the gradient of selection for an invading strategy, given a population state.
full_prob_increase_decrease_with_mutation	Calculates the probabilities of increasing/decreasing the frequency of a strategy in each possible population state.
gradient_selection	Calculates the gradient of selection given an invader and a resident strategy.
prob_increase_decrease	This function calculates for a given number of invaders the probability that the number increases or decreases with one.
prob_increase_decrease_with_mutation	This function calculates for a given number of invaders the probability that the number increases or decreases with taking into account a mutation rate.
transition_and_fixation_matrix	Calculates the transition matrix (only for the monomorphic states) and the fixation_probability probabilities.

__init__(*nb_strategies*, *payoffs*, *pop_size*, *group_size=2*, *mu=0*)

calculate_full_transition_matrix(*beta*, **args*)

Returns the full transition matrix in sparse representation.

Parameters

- **beta** (*float*) – Intensity of selection.
- **args** (*Optional[list]*) – Other arguments. Can be used to pass extra arguments to functions contained in the payoff matrix.

Returns The full transition matrix between the two strategies in sparse format.

Return type scipy.sparse.lil_matrix

calculate_stationary_distribution(*beta*, **args*)

Calculates the stationary distribution of the monomorphic states is mu = 0 (SML). Otherwise, it calculates the stationary distribution including all possible population states.

Parameters

- **beta** (*float*) – intensity of selection.
- **args** (*Optional[list]*) – extra arguments for calculating payoffs.

Returns A vector containing the stationary distribution

Return type numpy.ndarray

static fermi(*beta*, *fitness_diff*)

The fermi function determines the probability that the first type imitates the second.

Parameters

- **beta** (*float*) – intensity of selection
- **fitness_diff** (*float*) – Difference in fitneess between the strategies (f_a - f_b).

Returns the probability of imitation

Return type numpy.typing.ArrayLike

fitness_group(*x*, *i*, *j*, **args*)

In a population of x i-strategists and (Z-x) j strategists, where players interact in group of 'group_size' participants this function returns the average payoff of strategies i and j.

Parameters

- **x** (*int*) – number of individuals adopting strategy i in the population
- **i** (*int*) – index of strategy i
- **j** (*int*) – index of strategy j
- **args** (*Optional[list]*) – Other Parameters. This can be used to pass extra parameters to functions stored in the payoff matrix

Return type float

Returns

- *float*
- *Returns the difference in fitness between strategy i and j*

fitness_pair(*x*, *i*, *j*, **args*)

Calculates the fitness of strategy i versus strategy j, in a population of x i-strategists and (Z-x) j strategists, considering a 2-player game.

Parameters

- **x** (*int*) – number of i-strategists in the population
- **i** (*int*) – index of strategy i
- **j** (*int*) – index of strategy j
- **args** (*Optional[list]*) –

Return type float

Returns

- *float*

- *the fitness difference among the strategies*

fixation_probability(*invader*, *resident*, *beta*, **args*)

Function for calculating the fixation_probability probability of the invader in a population of residents.

Parameters

- **invader** (*int*) – index of the invading strategy
- **resident** (*int*) – index of the resident strategy
- **beta** (*float*) – intensity of selection
- **args** (*Optional[list]*) – Other arguments. Can be used to pass extra arguments to functions contained in the payoff matrix.

Returns The fixation_probability probability.

Return type float

See also:

egttools.numerical.PairwiseMoran

full_fitness_difference_group(*i*, *j*, *population_state*)

Calculate the fitness difference between strategies :param i and :param j assuming that player interact in groups of size N > 2 (n-player games).

Parameters

- **i** (*int*) – index of the strategy that will reproduce
- **j** (*int*) – index of the strategy that will die
- **population_state** (*numpy.ndarray[numpy.int64[m,1]]*) – vector containing the counts of each strategy in the population

Return type `float`

Returns

- *float*
- *The fitness difference between strategies i and j*

full_fitness_difference_pairwise(*i*, *j*, *population_state*)

Calculates the fitness of strategy i in a population with state :param population_state, assuming pairwise interactions (2-player game).

Parameters

- **i** (*int*) – index of the strategy that will reproduce
- **j** (*int*) – index of the strategy that will die
- **population_state** (*numpy.ndarray[numpy.int64[m,1]]*) – vector containing the counts of each strategy in the population

Return type `float`

Returns

- *float*
- *The fitness difference between the two strategies for the given population state*

full_gradient_selection(*population_state*, *beta*)

Calculates the gradient of selection for an invading strategy, given a population state.

Parameters

- **population_state** (*numpy.ndarray[np.int64[m,1]]*) – structure of unsigned integers containing the counts of each strategy in the population
- **beta** (*float*) – intensity of selection

Returns Matrix indicating the likelihood of change in the population given an starting point.

Return type numpy.ndarray[numpy.float64[m,m]]

full_prob_increase_decrease_with_mutation(*population_state*, *beta*)

Calculates the probabilities of increasing/decreasing the frequency of a strategy in each possible population state.

Parameters

- **population_state** (*numpy.ndarray[numpy.int64[m,1]]*) – structure of unsigned integers containing the counts of each strategy in the population
- **beta** (*float*) – intensity of selection

Returns

Returns an ndarray matrix with the probabilities of increasing/decreasing one individual adopting a given strategy in the current population state. All possible new state transition probabilities are returned.

e.g., given the population state (34, 33, 33) which represents a population of size 100 with 3 strategies - obviously the population state may be represented with a 2D tuple (34, 33) - then there are 6 possible changes to the population: (35, 32, 33), (35, 33, 32), (33, 34, 33), (33, 33, 34), (34, 34, 32), (34, 32, 34).

Return type numpy.ndarray[np.float64[m,m]]

gradient_selection(*k*, *invader*, *resident*, *beta*, **args*)

Calculates the gradient of selection given an invader and a resident strategy.

Parameters

- **k** (*int*) – number of invaders in the population
- **invader** (*int*) – index of the invading strategy
- **resident** (*int*) – index of the resident strategy
- **beta** (*float*) – intensity of selection
- **args** (*Optional[list[*) – other arguments. Can be used to pass extra arguments to functions contained in the payoff matrix.

Returns The gradient of selection.

Return type float

prob_increase_decrease(*k*, *invader*, *resident*, *beta*, **args*)

This function calculates for a given number of invaders the probability that the number increases or decreases with one.

Parameters

- **k** (*int*) – number of invaders in the population
- **invader** (*int*) – index of the invading strategy
- **resident** (*int*) – index of the resitent strategy

- **beta** (*float*) – intensity of selection
- **args** (*Optional[list]*) – other arguments. Can be used to pass extra arguments to functions contained in the payoff matrix.

Returns tuple(probability of increasing the number of invaders, probability of decreasing)

Return type Tuple[numpy.typing.ArrayLike, numpy.typing.ArrayLike]

prob_increase_decrease_with_mutation(*k*, *invader*, *resident*, *beta*, **args*)

This function calculates for a given number of invaders the probability that the number increases or decreases with taking into account a mutation rate.

Parameters

- **k** (*int*) – number of invaders in the population
- **invader** (*int*) – index of the invading strategy
- **resident** (*int*) – index of the resitent strategy
- **beta** (*float*) – intensity of selection
- **args** (*Optional[list]*) – other arguments. Can be used to pass extra arguments to functions contained in the payoff matrix.

Returns tuple(probability of increasing the number of invaders, probability of decreasing)

Return type Tuple[float, float]

transition_and_fixation_matrix(*beta*, **args*)

Calculates the transition matrix (only for the monomorphic states) and the fixation_probability probabilities.

This method calculates the transitions between monomorphic states. Thus, it assumes that we are in the small mutation limit (SML) of the moran process. Only use this method if this assumption is reasonable.

Parameters

- **beta** (*float*) – Intensity of selection.
- **args** (*Optional[list]*) – Other arguments. Can be used to pass extra arguments to functions contained in the payoff matrix.

Returns This method returns a tuple with the transition matrix as first element, and the matrix of fixation probabilities as second element.

Return type Tuple[numpy.ndarray[numpy.float64[m,m]], numpy.ndarray[numpy.float64[m,m]]]

egttools.analytical.sed_analytical.lil_matrix

class lil_matrix(*arg1*, *shape=None*, *dtype=None*, *copy=False*)

Bases: `scipy.sparse.base.spmatrix`, `scipy.sparse._index.IndexMixin`

Row-based list of lists sparse matrix

This is a structure for constructing sparse matrices incrementally. Note that inserting a single item can take linear time in the worst case; to construct a matrix efficiently, make sure the items are pre-sorted by index, per row.

This can be instantiated in several ways:

lil_matrix(D) with a dense matrix or rank-2 ndarray D

lil_matrix(S) with another sparse matrix S (equivalent to S.tolil())

lil_matrix((M, N), [dtype]) to construct an empty matrix with shape (M, N) dtype is optional, defaulting to dtype='d'.

dtype
Data type of the matrix

Type dtype

shape
Shape of the matrix

Type 2-tuple

ndim
Number of dimensions (this is always 2)

Type int

nnz
Number of stored values, including explicit zeros

data
LIL format data array of the matrix

rows
LIL format row index array of the matrix

Notes

Sparse matrices can be used in arithmetic operations: they support addition, subtraction, multiplication, division, and matrix power.

Advantages of the LIL format

- supports flexible slicing
- changes to the matrix sparsity structure are efficient

Disadvantages of the LIL format

- arithmetic operations LIL + LIL are slow (consider CSR or CSC)
- slow column slicing (consider CSC)
- slow matrix vector products (consider CSR or CSC)

Intended Usage

- LIL is a convenient format for constructing sparse matrices
- once a matrix has been constructed, convert to CSR or CSC format for fast arithmetic and matrix vector operations
- consider using the COO format when constructing large matrices

Data Structure

- An array (`self.rows`) of rows, each of which is a sorted list of column indices of non-zero elements.
- The corresponding nonzero values are stored in similar fashion in `self.data`.

Methods

asformat	Return this matrix in the passed format.
asfptype	Upcast matrix to a floating point format (if necessary)
astype	Cast the matrix elements to a specified type.
conj	Element-wise complex conjugation.
conjugate	Element-wise complex conjugation.
copy	Returns a copy of this matrix.
count_nonzero	Number of non-zero entries, equivalent to
diagonal	Returns the kth diagonal of the matrix.
dot	Ordinary dot product
getH	Return the Hermitian transpose of this matrix.
get_shape	Get shape of a matrix.
getcol	Returns a copy of column j of the matrix, as an (m x 1) sparse matrix (column vector).
getformat	Format of a matrix representation as a string.
getmaxprint	Maximum number of elements to display when printed.
getnnz	Number of stored values, including explicit zeros.
getrow	Returns a copy of the 'i'th row.
getrowview	Returns a view of the 'i'th row (without copying).
maximum	Element-wise maximum between this and another matrix.
mean	Compute the arithmetic mean along the specified axis.
minimum	Element-wise minimum between this and another matrix.
multiply	Point-wise multiplication by another matrix
nonzero	nonzero indices
power	Element-wise power.
reshape	Gives a new shape to a sparse matrix without changing its data.
resize	Resize the matrix in-place to dimensions given by shape
set_shape	See *reshape*.
setdiag	Set diagonal or off-diagonal elements of the array.
sum	Sum the matrix elements over a given axis.
toarray	Return a dense ndarray representation of this matrix.
tobsr	Convert this matrix to Block Sparse Row format.
tocoo	Convert this matrix to COOrdinate format.
tocsc	Convert this matrix to Compressed Sparse Column format.
tocsr	Convert this matrix to Compressed Sparse Row format.
todense	Return a dense matrix representation of this matrix.
todia	Convert this matrix to sparse DIAgonal format.
todok	Convert this matrix to Dictionary Of Keys format.
tolil	Convert this matrix to List of Lists format.
transpose	Reverses the dimensions of the sparse matrix.

Attributes

format	
ndim	
nnz	Number of stored values, including explicit zeros.
shape	Get shape of a matrix.

__abs__()

__add__(*other*)

__bool__()

__div__(*other*)

__eq__(*other*)
Return self==value.

__ge__(*other*)
Return self>=value.

__getattr__(*attr*)

__getitem__(*key*)

__gt__(*other*)
Return self>value.

__iadd__(*other*)

__idiv__(*other*)

__imul__(*other*)

__init__(*arg1*, *shape=None*, *dtype=None*, *copy=False*)

__isub__(*other*)

__iter__()

__itruediv__(*other*)

__le__(*other*)
Return self<=value.

__len__()

__lt__(*other*)
Return self<value.

__matmul__(*other*)

__mul__(*other*)
interpret other and call one of the following

self._mul_scalar() self._mul_vector() self._mul_multivector() self._mul_sparse_matrix()

__ne__(*other*)
Return self!=value.

__neg__()

__nonzero__()

__pow__(*other*)

__radd__(*other*)

__rdiv__(*other*)

__repr__()
Return repr(self).

__rmatmul__(*other*)

__rmul__(*other*)

__round__(*ndigits=0*)

__rsub__(*other*)

__rtruediv__(*other*)

__setitem__(*key*, *x*)

__str__()
Return str(self).

__sub__(*other*)

__truediv__(*other*)

asformat(*format*, *copy=False*)
Return this matrix in the passed format.

Parameters

- **format** (*{str, None}*) – The desired matrix format ("csr", "csc", "lil", "dok", "array", ...) or None for no conversion.
- **copy** (*bool, optional*) – If True, the result is guaranteed to not share data with self.

Returns A

Return type This matrix in the passed format.

asfptype()
Upcast matrix to a floating point format (if necessary)

astype(*dtype*, *casting='unsafe'*, *copy=True*)
Cast the matrix elements to a specified type.

Parameters

- **dtype** (*string or numpy dtype*) – Typecode or data-type to which to cast the data.
- **casting** (*{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional*) – Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility. 'no' means the data types should not be cast at all. 'equiv' means only byte-order changes are allowed. 'safe' means only casts which can preserve values are allowed. 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed. 'unsafe' means any data conversions may be done.
- **copy** (*bool, optional*) – If *copy* is `False`, the result might share some memory with this matrix. If *copy* is `True`, it is guaranteed that the result and this matrix do not share any memory.

conj(*copy=True*)
Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

Parameters **copy** (*bool, optional*) – If True, the result is guaranteed to not share data with self.

Returns **A**

Return type The element-wise complex conjugate.

conjugate(*copy=True*)

Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

Parameters **copy** (*bool, optional*) – If True, the result is guaranteed to not share data with self.

Returns **A**

Return type The element-wise complex conjugate.

copy()

Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

count_nonzero()

Number of non-zero entries, equivalent to

np.count_nonzero(a.toarray())

Unlike getnnz() and the nnz property, which return the number of stored entries (the length of the data attribute), this method counts the actual number of non-zero entries in data.

diagonal(*k=0*)

Returns the kth diagonal of the matrix.

Parameters **k** (*int, optional*) – Which diagonal to get, corresponding to elements a[i, i+k]. Default: 0 (the main diagonal).

New in version 1.0.

See also:

numpy.diagonal Equivalent numpy function.

Examples

```
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix([[1, 2, 0], [0, 0, 3], [4, 0, 5]])
>>> A.diagonal()
array([1, 0, 5])
>>> A.diagonal(k=1)
array([2, 3])
```

dot(*other*)

Ordinary dot product

Examples

```
>>> import numpy as np
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix([[1, 2, 0], [0, 0, 3], [4, 0, 5]])
>>> v = np.array([1, 0, -1])
>>> A.dot(v)
array([ 1, -3, -1], dtype=int64)
```

getH()

Return the Hermitian transpose of this matrix.

See also:

numpy.matrix.getH NumPy's implementation of *getH* for matrices

get_shape()

Get shape of a matrix.

getcol(*j*)

Returns a copy of column j of the matrix, as an (m x 1) sparse matrix (column vector).

getformat()

Format of a matrix representation as a string.

getmaxprint()

Maximum number of elements to display when printed.

getnnz(*axis=None*)

Number of stored values, including explicit zeros.

Parameters **axis** (*None, 0, or 1*) – Select between the number of values across the whole matrix, in each column, or in each row.

See also:

count_nonzero Number of non-zero entries

getrow(*i*)

Returns a copy of the 'i'th row.

getrowview(*i*)

Returns a view of the 'i'th row (without copying).

maximum(*other*)

Element-wise maximum between this and another matrix.

mean(*axis=None*, *dtype=None*, *out=None*)

Compute the arithmetic mean along the specified axis.

Returns the average of the matrix elements. The average is taken over all elements in the matrix by default, otherwise over the specified axis. `float64` intermediate and return values are used for integer inputs.

Parameters

- **axis** (*{-2, -1, 0, 1, None} optional*) – Axis along which the mean is computed. The default is to compute the mean of all elements in the matrix (i.e., `axis` = `None`).
- **dtype** (*data-type, optional*) – Type to use in computing the mean. For integer inputs, the default is `float64`; for floating point inputs, it is the same as the input dtype.

New in version 0.18.0.

- **out** (*np.matrix, optional*) – Alternative output matrix in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

  New in version 0.18.0.

Returns **m**

Return type np.matrix

See also:

`numpy.matrix.mean` NumPy's implementation of 'mean' for matrices

minimum(*other*)

Element-wise minimum between this and another matrix.

multiply(*other*)

Point-wise multiplication by another matrix

nonzero()

nonzero indices

Returns a tuple of arrays (row,col) containing the indices of the non-zero elements of the matrix.

Examples

```
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix([[1,2,0],[0,0,3],[4,0,5]])
>>> A.nonzero()
(array([0, 0, 1, 2, 2]), array([0, 1, 2, 0, 2]))
```

power(*n*, *dtype=None*)

Element-wise power.

reshape(*self*, *shape*, *order='C'*, *copy=False*)

Gives a new shape to a sparse matrix without changing its data.

Parameters

- **shape** (*length-2 tuple of ints*) – The new shape should be compatible with the original shape.
- **order** (*{'C', 'F'}, optional*) – Read the elements using this index order. 'C' means to read and write the elements using C-like index order; e.g., read entire first row, then second row, etc. 'F' means to read and write the elements using Fortran-like index order; e.g., read entire first column, then second column, etc.
- **copy** (*bool, optional*) – Indicates whether or not attributes of self should be copied whenever possible. The degree to which attributes are copied varies depending on the type of sparse matrix being used.

Returns **reshaped_matrix** – A sparse matrix with the given *shape*, not necessarily of the same format as the current object.

Return type sparse matrix

See also:

`numpy.matrix.reshape` NumPy's implementation of 'reshape' for matrices

resize(**shape*)

Resize the matrix in-place to dimensions given by `shape`

Any elements that lie within the new shape will remain at the same indices, while non-zero elements lying outside the new shape are removed.

Parameters **shape** (*(int, int)*) – number of rows and columns in the new matrix

Notes

The semantics are not identical to `numpy.ndarray.resize` or `numpy.resize`. Here, the same data will be maintained at each index before and after reshape, if that index is within the new bounds. In numpy, resizing maintains contiguity of the array, moving elements around in the logical matrix but not within a flattened representation.

We give no guarantees about whether the underlying data attributes (arrays, etc.) will be modified in place or replaced with new objects.

set_shape(*shape*)

See *reshape*.

setdiag(*values*, *k=0*)

Set diagonal or off-diagonal elements of the array.

Parameters

- **values** (*array_like*) – New values of the diagonal elements.

  Values may have any length. If the diagonal is longer than values, then the remaining diagonal entries will not be set. If values are longer than the diagonal, then the remaining values are ignored.

  If a scalar value is given, all of the diagonal is set to it.

- **k** (*int, optional*) – Which off-diagonal to set, corresponding to elements a[i,i+k]. Default: 0 (the main diagonal).

sum(*axis=None*, *dtype=None*, *out=None*)

Sum the matrix elements over a given axis.

Parameters

- **axis** (*{-2, -1, 0, 1, None} optional*) – Axis along which the sum is computed. The default is to compute the sum of all the matrix elements, returning a scalar (i.e., `axis = None`).

- **dtype** (*dtype, optional*) – The type of the returned matrix and of the accumulator in which the elements are summed. The dtype of `a` is used by default unless `a` has an integer dtype of less precision than the default platform integer. In that case, if `a` is signed then the platform integer is used while if `a` is unsigned then an unsigned integer of the same precision as the platform integer is used.

  New in version 0.18.0.

- **out** (*np.matrix, optional*) – Alternative output matrix in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

  New in version 0.18.0.

Returns **sum_along_axis** – A matrix with the same shape as `self`, with the specified axis removed.

Return type np.matrix

See also:

`numpy.matrix.sum` NumPy's implementation of 'sum' for matrices

`toarray`(*order=None*, *out=None*)

Return a dense ndarray representation of this matrix.

Parameters

- **order** (*{'C', 'F'}, optional*) – Whether to store multidimensional data in C (row-major) or Fortran (column-major) order in memory. The default is 'None', indicating the NumPy default of C-ordered. Cannot be specified in conjunction with the `out` argument.
- **out** (*ndarray, 2-D, optional*) – If specified, uses this array as the output buffer instead of allocating a new array to return. The provided array must have the same shape and dtype as the sparse matrix on which you are calling the method. For most sparse types, `out` is required to be memory contiguous (either C or Fortran ordered).

Returns **arr** – An array with the same shape and containing the same data represented by the sparse matrix, with the requested memory order. If `out` was passed, the same object is returned after being modified in-place to contain the appropriate values.

Return type ndarray, 2-D

`tobsr`(*blocksize=None*, *copy=False*)

Convert this matrix to Block Sparse Row format.

With copy=False, the data/indices may be shared between this matrix and the resultant bsr_matrix.

When blocksize=(R, C) is provided, it will be used for construction of the bsr_matrix.

`tocoo`(*copy=False*)

Convert this matrix to COOrdinate format.

With copy=False, the data/indices may be shared between this matrix and the resultant coo_matrix.

`tocsc`(*copy=False*)

Convert this matrix to Compressed Sparse Column format.

With copy=False, the data/indices may be shared between this matrix and the resultant csc_matrix.

`tocsr`(*copy=False*)

Convert this matrix to Compressed Sparse Row format.

With copy=False, the data/indices may be shared between this matrix and the resultant csr_matrix.

`todense`(*order=None*, *out=None*)

Return a dense matrix representation of this matrix.

Parameters

- **order** (*{'C', 'F'}, optional*) – Whether to store multi-dimensional data in C (row-major) or Fortran (column-major) order in memory. The default is 'None', indicating the NumPy default of C-ordered. Cannot be specified in conjunction with the `out` argument.
- **out** (*ndarray, 2-D, optional*) – If specified, uses this array (or `numpy.matrix`) as the output buffer instead of allocating a new array to return. The provided array must have the same shape and dtype as the sparse matrix on which you are calling the method.

Returns **arr** – A NumPy matrix object with the same shape and containing the same data represented by the sparse matrix, with the requested memory order. If `out` was passed and was an array (rather than a `numpy.matrix`), it will be filled with the appropriate values and returned wrapped in a `numpy.matrix` object that shares the same memory.

Return type numpy.matrix, 2-D

todia(*copy=False*)

Convert this matrix to sparse DIAgonal format.

With copy=False, the data/indices may be shared between this matrix and the resultant dia_matrix.

todok(*copy=False*)

Convert this matrix to Dictionary Of Keys format.

With copy=False, the data/indices may be shared between this matrix and the resultant dok_matrix.

tolil(*copy=False*)

Convert this matrix to List of Lists format.

With copy=False, the data/indices may be shared between this matrix and the resultant lil_matrix.

transpose(*axes=None*, *copy=False*)

Reverses the dimensions of the sparse matrix.

Parameters

- **axes** (*None, optional*) – This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value.
- **copy** (*bool, optional*) – Indicates whether or not attributes of `self` should be copied whenever possible. The degree to which attributes are copied varies depending on the type of sparse matrix being used.

Returns **p**

Return type `self` with the dimensions reversed.

See also:

`numpy.matrix.transpose` NumPy's implementation of 'transpose' for matrices

__array_priority__ = 10.1

__hash__ = None

format = 'lil'

ndim = 2

property nnz

Number of stored values, including explicit zeros.

See also:

count_nonzero Number of non-zero entries

property shape

Get shape of a matrix.

2.7 egttools.behaviors

API reference documentation for `behaviors` submodule.

egttools.behaviors.CRD	API reference documentation for `behaviors.CRD` submodule.
egttools.behaviors.NormalForm	API reference documentation for `behaviors.NormalForm` submodule.
egttools.behaviors.pgg_behaviors	The `behaviors.pgg_behaviors` submodule contains strategies which can be used with *egttools.games.pgg* game.

2.7.1 egttools.behaviors.CRD

API reference documentation for `behaviors.CRD` submodule.

Classes

AbstractCRDStrategy	
CRDMemoryOnePlayer	This strategy contributes in function of the contributions of the rest of the group in the previous round.
MovingAverageCRDStrategy	

egttools.behaviors.CRD.AbstractCRDStrategy

class AbstractCRDStrategy(*self:* egttools.numerical.behaviors.CRD.AbstractCRDStrategy) → None

Bases: `pybind11_builtins.pybind11_object`

Methods

get_action	Returns an action in function of time_step round and the previous action action_prev of the opponent.
type	Returns a string indicating the Strategy Type.

__init__(*self:* egttools.numerical.behaviors.CRD.AbstractCRDStrategy) → None

__new__(***kwargs*)

get_action(*self:* egttools.numerical.behaviors.CRD.AbstractCRDStrategy, *time_step:* *int*, *group_contributions_prev:* *int*) → int

Returns an action in function of time_step round and the previous action action_prev of the opponent.

Parameters

- **time_step** (*int*) – Current round.
- **group_contributions_prev** (*int*) – Sum of contributions of the other members of the group (excluding the focal player) in the previous round.

Returns The action selected by the strategy.

Return type int

See also:

egttools.behaviors.CRD.CRDMemoryOnePlayer, egttools.behaviors.NormalForm.TwoActions.Cooperator, egttools.behaviors.NormalForm.TwoActions.Defector, egttools.behaviors.NormalForm.TwoActions.Random, egttools.behaviors.NormalForm.TwoActions.TFT, egttools.behaviors.NormalForm.TwoActions.SuspiciousTFT, egttools.behaviors.NormalForm.TwoActions.GenerousTFT, egttools.behaviors.NormalForm.TwoActions.GradualTFT, egttools.behaviors.NormalForm.TwoActions.ImperfectTFT, egttools.behaviors.NormalForm.TwoActions.TFTT, egttools.behaviors.NormalForm.TwoActions.TTFT, egttools.behaviors.NormalForm.TwoActions.GRIM, egttools.behaviors.NormalForm.TwoActions.Pavlov

type(*self:* egttools.numerical.behaviors.CRD.AbstractCRDStrategy) → str

Returns a string indicating the Strategy Type.

egttools.behaviors.CRD.CRDMemoryOnePlayer

class CRDMemoryOnePlayer(*self:* egttools.numerical.behaviors.CRD.CRDMemoryOnePlayer, *personal_threshold: int*, *initial_action: int*, *action_above: int*, *action_equal: int*, *action_below: int*) → None

Bases: *egttools.numerical.behaviors.CRD.AbstractCRDStrategy*

This strategy contributes in function of the contributions of the rest of the group in the previous round.

This strategy contributes @param initial_action in the first round of the game, afterwards compares the sum of contributions of the other members of the group in the previous round (a_{-i}(t-1)) to a :param personal_threshold. If the a_{-i}(t-1)) > personal_threshold the agent contributions :param action_above, if a_{-i}(t-1)) = personal_threshold it contributes :param action_equal or if a_{-i}(t-1)) < personal_threshold it contributes :param action_below.

Parameters

- **personal_threshold** (*int*) – threshold value compared to the contributions of the other members of the group
- **initial_action** (*int*) – Contribution in the first round
- **action_above** (*int*) – contribution if a_{-i}(t-1)) > personal_threshold
- **action_equal** (*int*) – contribution if a_{-i}(t-1)) = personal_threshold
- **action_below** (*int*) – contribution if a_{-i}(t-1)) < personal_threshold

See also:

egttools.behaviors.AbstractGame

Methods

get_action	Returns an action in function of time_step round and the previous action action_prev of the opponent.
type	Returns a string indicating the Strategy Type.

__init__(*self:* egttools.numerical.behaviors.CRD.CRDMemoryOnePlayer, *personal_threshold:* *int*, *initial_action:* *int*, *action_above:* *int*, *action_equal:* *int*, *action_below:* *int*) → None

This strategy contributes in function of the contributions of the rest of the group in the previous round.

This strategy contributes @param initial_action in the first round of the game, afterwards compares the sum of contributions of the other members of the group in the previous round (a_{-i}(t-1)) to a :param personal_threshold. If the a_{-i}(t-1)) > personal_threshold the agent contributions :param action_above, if a_{-i}(t-1)) = personal_threshold it contributes :param action_equal or if a_{-i}(t-1)) < personal_threshold it contributes :param action_below.

Parameters

- **personal_threshold** (*int*) – threshold value compared to the contributions of the other members of the group
- **initial_action** (*int*) – Contribution in the first round
- **action_above** (*int*) – contribution if a_{-i}(t-1)) > personal_threshold
- **action_equal** (*int*) – contribution if a_{-i}(t-1)) = personal_threshold
- **action_below** (*int*) – contribution if a_{-i}(t-1)) < personal_threshold

See also:

`egttools.behaviors.AbstractGame`

__new__(***kwargs*)

__str__(*self:* egttools.numerical.behaviors.CRD.CRDMemoryOnePlayer) → str

get_action(*self:* egttools.numerical.behaviors.CRD.CRDMemoryOnePlayer, *time_step:* *int*, *group_contributions_prev:* *int*) → int

Returns an action in function of time_step round and the previous action action_prev of the opponent.

Parameters

- **time_step** (*int*) – Current round.
- **group_contributions_prev** (*int*) – Sum of contributions of the other members of the group (without the focal player) in the previous round.

Returns The action selected by the strategy.

Return type int

See also:

`egttools.behaviors.AbstractGame`, *`egttools.games.AbstractGame`*, `egttools.behaviors.NormalForm.TwoActions.Cooperator`, `egttools.behaviors.NormalForm.TwoActions.Defector`, `egttools.behaviors.NormalForm.TwoActions.Random`, `egttools.behaviors.NormalForm.TwoActions.TFT`, `egttools.behaviors.NormalForm.TwoActions.SuspiciousTFT`, `egttools.behaviors.NormalForm.TwoActions.GenerousTFT`, `egttools.behaviors.NormalForm.TwoActions.GradualTFT`, `egttools.behaviors.NormalForm.TwoActions.ImperfectTFT`, `egttools.behaviors.NormalForm.TwoActions.TFTT`, `egttools.behaviors.NormalForm.TwoActions.TFTT`, `egttools.behaviors.NormalForm.TwoActions.GRIM`

type(*self:* egttools.numerical.behaviors.CRD.CRDMemoryOnePlayer) → str
Returns a string indicating the Strategy Type.

egttools.behaviors.CRD.MovingAverageCRDStrategy

class MovingAverageCRDStrategy(*a0*, *aa*, *am*, *ab*, *group_size*)
Bases: *egttools.numerical.behaviors.CRD.AbstractCRDStrategy*

Methods

get_action	Returns an action in function of time_step round and the previous action action_prev of the opponent.
type	Returns a string indicating the Strategy Type.

__init__(*a0*, *aa*, *am*, *ab*, *group_size*)

__new__(***kwargs*)

__str__()
Return str(self).

get_action(*time_step*, *group_contributions_prev*)
Returns an action in function of time_step round and the previous action action_prev of the opponent.

Parameters

- **time_step** (*int*) – Current round.
- **group_contributions_prev** (*int*) – Sum of contributions of the other members of the group (excluding the focal player) in the previous round.

Returns The action selected by the strategy.

Return type int

See also:

egttools.behaviors.CRD.CRDMemoryOnePlayer, egttools.behaviors.NormalForm.TwoActions.Cooperator, egttools.behaviors.NormalForm.TwoActions.Defector, egttools.behaviors.NormalForm.TwoActions.Random, egttools.behaviors.NormalForm.TwoActions.TFT, egttools.behaviors.NormalForm.TwoActions.SuspiciousTFT, egttools.behaviors.NormalForm.TwoActions.GenerousTFT, egttools.behaviors.NormalForm.TwoActions.GradualTFT, egttools.behaviors.NormalForm.TwoActions.ImperfectTFT, egttools.behaviors.NormalForm.TwoActions.TFTT, egttools.behaviors.NormalForm.TwoActions.TTFT, egttools.behaviors.NormalForm.TwoActions.GRIM, egttools.behaviors.NormalForm.TwoActions.Pavlov

type(*self:* egttools.numerical.behaviors.CRD.AbstractCRDStrategy) → str
Returns a string indicating the Strategy Type.

egttools.behaviors.CRD.moving_average

egttools.behaviors.CRD.moving_average

Classes

AbstractCRDStrategy
MovingAverageCRDStrategy

egttools.behaviors.CRD.moving_average.AbstractCRDStrategy

class AbstractCRDStrategy(*self:* egttools.numerical.behaviors.CRD.AbstractCRDStrategy) → None

Bases: `pybind11_builtins.pybind11_object`

Methods

get_action	Returns an action in function of time_step round and the previous action action_prev of the opponent.
type	Returns a string indicating the Strategy Type.

__init__(*self:* egttools.numerical.behaviors.CRD.AbstractCRDStrategy) → None

__new__(***kwargs*)

get_action(*self:* egttools.numerical.behaviors.CRD.AbstractCRDStrategy, *time_step:* *int*, *group_contributions_prev:* *int*) → int

Returns an action in function of time_step round and the previous action action_prev of the opponent.

Parameters

- **time_step** (*int*) – Current round.
- **group_contributions_prev** (*int*) – Sum of contributions of the other members of the group (excluding the focal player) in the previous round.

Returns The action selected by the strategy.

Return type int

See also:

egttools.behaviors.CRD.CRDMemoryOnePlayer, egttools.behaviors.NormalForm.TwoActions.Cooperator, egttools.behaviors.NormalForm.TwoActions.Defector, egttools.behaviors.NormalForm.TwoActions.Random, egttools.behaviors.NormalForm.TwoActions.TFT, egttools.behaviors.NormalForm.TwoActions.SuspiciousTFT, egttools.behaviors.NormalForm.TwoActions.GenerousTFT, egttools.behaviors.NormalForm.TwoActions.GradualTFT, egttools.behaviors.NormalForm.TwoActions.ImperfectTFT, egttools.behaviors.NormalForm.TwoActions.TFTT, egttools.behaviors.NormalForm.TwoActions.TTFT, egttools.behaviors.NormalForm.TwoActions.GRIM, egttools.behaviors.NormalForm.TwoActions.Pavlov

type(*self:* egttools.numerical.behaviors.CRD.AbstractCRDStrategy) → str

Returns a string indicating the Strategy Type.

egttools.behaviors.CRD.moving_average.MovingAverageCRDStrategy

class MovingAverageCRDStrategy(*a0*, *aa*, *am*, *ab*, *group_size*)

Bases: *egttools.numerical.behaviors.CRD.AbstractCRDStrategy*

Methods

get_action	Returns an action in function of time_step round and the previous action action_prev of the opponent.
type	Returns a string indicating the Strategy Type.

__init__(*a0*, *aa*, *am*, *ab*, *group_size*)

__new__(**kwargs*)

__str__()

Return str(self).

get_action(*time_step*, *group_contributions_prev*)

Returns an action in function of time_step round and the previous action action_prev of the opponent.

Parameters

- **time_step** (*int*) – Current round.
- **group_contributions_prev** (*int*) – Sum of contributions of the other members of the group (excluding the focal player) in the previous round.

Returns The action selected by the strategy.

Return type int

See also:

egttools.behaviors.CRD.CRDMemoryOnePlayer, egttools.behaviors.NormalForm.TwoActions.Cooperator, egttools.behaviors.NormalForm.TwoActions.Defector, egttools.behaviors.NormalForm.TwoActions.Random, egttools.behaviors.NormalForm.TwoActions.TFT, egttools.behaviors.NormalForm.TwoActions.SuspiciousTFT, egttools.behaviors.NormalForm.TwoActions.GenerousTFT, egttools.behaviors.NormalForm.TwoActions.GradualTFT, egttools.behaviors.NormalForm.TwoActions.ImperfectTFT, egttools.behaviors.NormalForm.TwoActions.TFTT, egttools.behaviors.NormalForm.TwoActions.TTFT, egttools.behaviors.NormalForm.TwoActions.GRIM, egttools.behaviors.NormalForm.TwoActions.Pavlov

type(*self:* egttools.numerical.behaviors.CRD.AbstractCRDStrategy) → str

Returns a string indicating the Strategy Type.

2.7.2 egttools.behaviors.NormalForm

API reference documentation for `behaviors.NormalForm` submodule.

Classes

AbstractNFGStrategy

egttools.behaviors.NormalForm.AbstractNFGStrategy

class AbstractNFGStrategy(*self: egttools.numerical.behaviors.NormalForm.AbstractNFGStrategy*) → None

Bases: `pybind11_builtins.pybind11_object`

Methods

get_action	Returns an action in function of time_step round and the previous action action_prev of the opponent.
is_stochastic	Property indicating if the strategy is stochastic.
type	Returns a string indicating the Strategy Type.

__init__(*self: egttools.numerical.behaviors.NormalForm.AbstractNFGStrategy*) → None

__new__(***kwargs*)

get_action(*self: egttools.numerical.behaviors.NormalForm.AbstractNFGStrategy*, *time_step: int*, *action_prev: int*) → int

Returns an action in function of time_step round and the previous action action_prev of the opponent.

Parameters

- **time_step** (*int*) – Current round.
- **action_prev** (*int*) – Previous action of the opponent.

Returns The action selected by the strategy.

Return type int

See also:

`egttools.behaviors.NormalForm.TwoActions.Cooperator`, `egttools.behaviors.NormalForm.TwoActions.Defector`, `egttools.behaviors.NormalForm.TwoActions.Random`, `egttools.behaviors.NormalForm.TwoActions.TFT`, `egttools.behaviors.NormalForm.TwoActions.SuspiciousTFT`, `egttools.behaviors.NormalForm.TwoActions.GenerousTFT`, `egttools.behaviors.NormalForm.TwoActions.GradualTFT`, `egttools.behaviors.NormalForm.TwoActions.ImperfectTFT`, `egttools.behaviors.NormalForm.TwoActions.TFTT`, `egttools.behaviors.NormalForm.TwoActions.TTFT`, `egttools.behaviors.NormalForm.TwoActions.GRIM`, `egttools.behaviors.NormalForm.TwoActions.Pavlov`

is_stochastic(*self: egttools.numerical.behaviors.NormalForm.AbstractNFGStrategy*) → bool

Property indicating if the strategy is stochastic.

type(*self: egttools.numerical.behaviors.NormalForm.AbstractNFGStrategy*) → str

Returns a string indicating the Strategy Type.

`egttools.behaviors.NormalForm.TwoActions`	API reference documentation for `behaviors.NormalForm.TwoActions` submodule.

2.7.3 egttools.behaviors.pgg_behaviors

The `behaviors.pgg_behaviors` submodule contains strategies which can be used with *egttools.games.pgg* game.

Functions

player_factory	**rtype** `List`[*PGGOneShotStrategy*]

egttools.behaviors.pgg_behaviors.player_factory

player_factory(*actions*)

Return type `List`[*PGGOneShotStrategy*]

Classes

PGGOneShotStrategy

egttools.behaviors.pgg_behaviors.PGGOneShotStrategy

class PGGOneShotStrategy(*action*)

Bases: `object`

Methods

get_action	**rtype** `int`

Attributes

type	**rtype** str

__init__(*action*)

get_action()

Return type int

property type: str

Return type str

2.8 egttools.games

API reference documentation for the `games` submodule.

Classes

AbstractGame	
CRDGame	Collective Risk Dilemma. This allows you to define any number of strategies by passing them as a list. All strategies must be of type AbstractCRDStrategy *.
CRDGameTU	Collective Risk Dilemma. This allows you to define any number of strategies by passing them as a list. All strategies must be of type AbstractCRDStrategy *.
InformalRiskGame	This game has been taken from the model introduced in
NormalFormGame	Overloaded function.
PGG	

2.8.1 egttools.games.AbstractGame

class AbstractGame(*self:* egttools.numerical.games.AbstractGame) → None

Bases: `pybind11_builtins.pybind11_object`

Methods

calculate_fitness	Estimates the fitness for a player_type in the population with state :param strategies.
calculate_payoffs	Estimates the payoffs for each strategy and returns the values in a matrix.
nb_strategies	Number of different strategies playing the game.
payoff	Returns the payoff of a strategy given a group composition.
payoffs	returns the payoff matrix of the game.
play	Updates the vector of payoffs with the payoffs of each player after playing the game.
save_payoffs	Stores the payoff matrix in a txt file.
type	returns the type of game.

__init__(*self:* egttools.numerical.games.AbstractGame) → None

__new__(***kwargs*)

__str__(*self:* egttools.numerical.games.AbstractGame) → str

calculate_fitness(*self:* egttools.numerical.games.AbstractGame, *player_type: int*, *pop_size: int*, *strategies: numpy.ndarray[numpy.uint64[m, 1]]*) → float

Estimates the fitness for a player_type in the population with state :param strategies.

This function assumes that the player with strategy player_type is not included in the vector of strategy counts strategies.

Parameters

- **player_type** (*int*) – The index of the strategy used by the player.
- **pop_size** (*int*) – The size of the population.
- **strategies** (*numpy.ndarray[numpy.uint64[m, 1]]*) – A vector of counts of each strategy. The current state of the population.

Returns The fitness of the strategy in the population state given by strategies.

Return type float

calculate_payoffs(*self:* egttools.numerical.games.AbstractGame) → numpy.ndarray[numpy.float64[m, n]]

Estimates the payoffs for each strategy and returns the values in a matrix. Each row of the matrix represents a strategy and each column a game state. E.g., in case of a 2 player game, each entry a_ij gives the payoff for strategy i against strategy j. In case of a group game, each entry a_ij gives the payoff of strategy i for game state j, which represents the group composition.

Returns A matrix with the expected payoffs for each strategy given each possible game state.

Return type numpy.ndarray[numpy.float64[m, n]]

nb_strategies(*self:* egttools.numerical.games.AbstractGame) → int

Number of different strategies playing the game.

payoff(*self:* egttools.numerical.games.AbstractGame, *strategy: int*, *group_composition: List[int]*) → float

Returns the payoff of a strategy given a group composition.

If the group composition does not include the strategy, the payoff should be zero.

Parameters

- **strategy** (*int*) – The index of the strategy used by the player.
- **group_composition** (*List[int]*) – List with the group composition. The structure of this list depends on the particular implementation of this abstract method.

Returns The payoff value.

Return type float

payoffs(*self:* egttools.numerical.games.AbstractGame) → numpy.ndarray[numpy.float64[m, n]]
returns the payoff matrix of the game.

play(*self:* egttools.numerical.games.AbstractGame, *group_composition: List[int]*, *game_payoffs: List[float]*) → None
Updates the vector of payoffs with the payoffs of each player after playing the game.

This method will run the game using the players and player types defined in :param group_composition, and will update the vector :param game_payoffs with the resulting payoff of each player.

Parameters

- **group_composition** (*List[int]*) – A list with counts of the number of players of each strategy in the group.
- **game_payoffs** (*List[float]*) – A list used as container for the payoffs of each player

save_payoffs(*self:* egttools.numerical.games.AbstractGame, *file_name: str*) → None
Stores the payoff matrix in a txt file.

Parameters **file_name** (*str*) – Name of the file in which the data will be stored.

type(*self:* egttools.numerical.games.AbstractGame) → str
returns the type of game.

2.8.2 egttools.games.CRDGame

class CRDGame(*self:* egttools.numerical.games.CRDGame, *endowment: int*, *threshold: int*, *nb_rounds: int*, *group_size: int*, *risk: float*, *strategies: list*) → None
Bases: *egttools.numerical.games.AbstractGame*

Collective Risk Dilemma. This allows you to define any number of strategies by passing them as a list. All strategies must be of type AbstractCRDStrategy *.

The CRD dilemma implemented here follows the description of: Milinski, M., Sommerfeld, R. D., Krambeck, H.-J., Reed, F. A., & Marotzke, J. (2008). The collective-risk social dilemma and the prevention of simulated dangerous climate change. Proceedings of the National Academy of Sciences of the United States of America, 105(7), 2291–2294. https://doi.org/10.1073/pnas.0709546105

Parameters

- **endowment** (*int*) – Initial endowment for all players.
- **threshold** (*int*) – Collective target that the group must reach.
- **nb_rounds** (*int*) – Number of rounds of the game.
- **group_size** (*int*) – Size of the group that will play the CRD.
- **risk** (*float*) – The probability that all members will lose their remaining endowment if the threshold is not achieved.
- **strategies** (*List[egttools.behaviors.AbstractCRDStrategy]*) – A list containing references of AbstractCRDStrategy strategies (or child classes).

See also:

egttools.games.AbstractGame, *egttools.games.NormalFormGame*

Methods

calculate_fitness	calculates the fitness of an individual of a given strategy given a population state.It always assumes that the population state does not contain the current individual
calculate_group_achievement	calculates the group achievement for a given stationary distribution.
calculate_payoffs	updates the internal payoff and coop_level matrices by calculating the payoff of each strategy given any possible strategy pair
calculate_polarization	calculates the fraction of players that contribute above, below or equal to the fair contribution (E/2)in a give population state.
calculate_polarization_success	calculates the fraction of players (from successful groups)) that contribute above, below or equal to the fair contribution (E/2)in a give population state.
calculate_population_group_achievement	calculates the group achievement in the population at a given state.
nb_strategies	Number of different strategies which are playing the game.
payoff	returns the payoff of a strategy given a group composition.
payoffs	returns the expected payoffs of each strategy vs each possible game state
play	
save_payoffs	Saves the payoff matrix in a txt file.
type	

Attributes

endowment	Initial endowment for all players.
group_achievement_per_group	
group_size	Size of the group which will play the game.
nb_rounds	Number of rounds of the game.
nb_states	Number of combinations of 2 strategies that can be matched in the game.
risk	Probability that all players will lose their remaining endowment if the target si not achieved.
strategies	A list with pointers to the strategies that are playing the game.
target	Collective target which needs to be achieved by the group.

__init__(*self:* egttools.numerical.games.CRDGame, *endowment: int*, *threshold: int*, *nb_rounds: int*, *group_size: int*, *risk: float*, *strategies: list*) → None

Collective Risk Dilemma. This allows you to define any number of strategies by passing them as a list. All strategies must be of type AbstractCRDStrategy *.

The CRD dilemma implemented here follows the description of: Milinski, M., Sommerfeld, R. D., Krambeck, H.-J., Reed, F. A., & Marotzke, J. (2008). The collective-risk social dilemma and the prevention of simulated dangerous climate change. Proceedings of the National Academy of Sciences of the United States of America, 105(7), 2291–2294. https://doi.org/10.1073/pnas.0709546105

Parameters

- **endowment** (*int*) – Initial endowment for all players.
- **threshold** (*int*) – Collective target that the group must reach.
- **nb_rounds** (*int*) – Number of rounds of the game.
- **group_size** (*int*) – Size of the group that will play the CRD.
- **risk** (*float*) – The probability that all members will lose their remaining endowment if the threshold is not achieved.
- **strategies** (*List[egttools.behaviors.AbstractCRDStrategy]*) – A list containing references of AbstractCRDStrategy strategies (or child classes).

See also:

egttools.games.AbstractGame, *egttools.games.NormalFormGame*

__new__(***kwargs*)

__str__(*self:* egttools.numerical.games.CRDGame) → str

calculate_fitness(*self:* egttools.numerical.games.CRDGame, *player_strategy: int*, *pop_size: int*, *population_state: numpy.ndarray[numpy.uint64[m, 1]]*) → float

calculates the fitness of an individual of a given strategy given a population state.It always assumes that the population state does not contain the current individual

calculate_group_achievement(*self:* egttools.numerical.games.CRDGame, *population_size: int*, *stationary_distribution: numpy.ndarray[numpy.float64[m, 1]]*) → float

calculates the group achievement for a given stationary distribution.

calculate_payoffs(*self:* egttools.numerical.games.CRDGame) → numpy.ndarray[numpy.float64[m, n]]

updates the internal payoff and coop_level matrices by calculating the payoff of each strategy given any possible strategy pair

calculate_polarization(*self:* egttools.numerical.games.CRDGame, *population_size: int*, *population_state: numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[3, 1]]

calculates the fraction of players that contribute above, below or equal to the fair contribution (E/2)in a give population state.

calculate_polarization_success(*self:* egttools.numerical.games.CRDGame, *population_size: int*, *population_state: numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[3, 1]]

calculates the fraction of players (from successful groups)) that contribute above, below or equal to the fair contribution (E/2)in a give population state.

calculate_population_group_achievement(*self:* egttools.numerical.games.CRDGame, *population_size: int*, *population_state: numpy.ndarray[numpy.uint64[m, 1]]*) → float

calculates the group achievement in the population at a given state.

nb_strategies(*self:* egttools.numerical.games.CRDGame) → int
Number of different strategies which are playing the game.

payoff(*self: egttools.numerical.games.CRDGame, strategy: int, strategy pair: List[int]*) → float
returns the payoff of a strategy given a group composition.

payoffs(*self:* egttools.numerical.games.CRDGame) → numpy.ndarray[numpy.float64[m, n]]
returns the expected payoffs of each strategy vs each possible game state

play(*self:* egttools.numerical.games.CRDGame, *arg0: List[int], arg1: List[float]*) → None

save_payoffs(*self:* egttools.numerical.games.CRDGame, *arg0: str*) → None
Saves the payoff matrix in a txt file.

type(*self:* egttools.numerical.games.CRDGame) → str

property endowment
Initial endowment for all players.

property group_achievement_per_group

property group_size
Size of the group which will play the game.

property nb_rounds
Number of rounds of the game.

property nb_states
Number of combinations of 2 strategies that can be matched in the game.

property risk
Probability that all players will lose their remaining endowment if the target si not achieved.

property strategies
A list with pointers to the strategies that are playing the game.

property target
Collective target which needs to be achieved by the group.

2.8.3 egttools.games.CRDGameTU

class CRDGameTU(*self:* egttools.numerical.games.CRDGameTU, *endowment: int, threshold: int, min_rounds: int, group_size: int, risk: float, tu: egttools.numerical.distributions.TimingUncertainty, strategies: list*) → None

Bases: *egttools.numerical.games.AbstractGame*

Collective Risk Dilemma. This allows you to define any number of strategies by passing them as a list. All strategies must be of type AbstractCRDStrategy *.

The CRD dilemma implemented here follows the description of: Milinski, M., Sommerfeld, R. D., Krambeck, H.-J., Reed, F. A., & Marotzke, J. (2008). The collective-risk social dilemma and the prevention of simulated dangerous climate change. Proceedings of the National Academy of Sciences of the United States of America, 105(7), 2291–2294. https://doi.org/10.1073/pnas.0709546105

Parameters

- **endowment** (*int*) – Initial endowment for all players.
- **threshold** (*int*) – Collective target that the group must reach.
- **min_rounds** (*int*) – Minimum number of rounds of the game.
- **group_size** (*int*) – Size of the group that will play the CRD.

- **risk** (*float*) – The probability that all members will lose their remaining endowment if the threshold is not achieved.
- **tu** (*egttools.distributions.TimingUncertainty*) – Timing uncertainty object which can output the total number of rounds of the game according to pre-defined distribution (by default it uses a geometric distribution).
- **strategies** (*List[egttools.behaviors.AbstractCRDStrategy]*) – A list containing references of AbstractCRDStrategy strategies (or child classes).

See also:

egttools.games.AbstractGame, *egttools.games.NormalFormGame*, *egttools.games.CRDGame*

Methods

calculate_fitness	calculates the fitness of an individual of a given strategy given a population state.It always assumes that the population state does not contain the current individual
calculate_group_achievement	calculates the group achievement for a given stationary distribution.
calculate_payoffs	updates the internal payoff and coop_level matrices by calculating the payoff of each strategy given any possible strategy pair
calculate_polarization	calculates the fraction of players that contribute above, below or equal to the fair contribution (E/2)in a give population state.
calculate_polarization_success	calculates the fraction of players (from successful groups)) that contribute above, below or equal to the fair contribution (E/2)in a give population state.
calculate_population_group_achievement	calculates the group achievement in the population at a given state.
nb_strategies	Number of different strategies which are playing the game.
payoff	returns the payoff of a strategy given a group composition.
payoffs	returns the expected payoffs of each strategy vs each possible game state
play	
save_payoffs	Saves the payoff matrix in a txt file.
type	

Attributes

endowment	Initial endowment for all players.
group_size	Size of the group which will play the game.
min_rounds	Minimum number of rounds of the game.
nb_states	Number of combinations of 2 strategies that can be matched in the game.
risk	Probability that all players will lose their remaining endowment if the target si not achieved.
strategies	A list with pointers to the strategies that are playing the game.
target	Collective target which needs to be achieved by the group.

__init__(*self:* egttools.numerical.games.CRDGameTU, *endowment: int*, *threshold: int*, *min_rounds: int*, *group_size: int*, *risk: float*, *tu: egttools.numerical.distributions.TimingUncertainty*, *strategies: list*) → None

Collective Risk Dilemma. This allows you to define any number of strategies by passing them as a list. All strategies must be of type AbstractCRDStrategy *.

The CRD dilemma implemented here follows the description of: Milinski, M., Sommerfeld, R. D., Krambeck, H.-J., Reed, F. A., & Marotzke, J. (2008). The collective-risk social dilemma and the prevention of simulated dangerous climate change. Proceedings of the National Academy of Sciences of the United States of America, 105(7), 2291–2294. https://doi.org/10.1073/pnas.0709546105

Parameters

- **endowment** (*int*) – Initial endowment for all players.
- **threshold** (*int*) – Collective target that the group must reach.
- **min_rounds** (*int*) – Minimum number of rounds of the game.
- **group_size** (*int*) – Size of the group that will play the CRD.
- **risk** (*float*) – The probability that all members will lose their remaining endowment if the threshold is not achieved.
- **tu** (*egttools.distributions.TimingUncertainty*) – Timing uncertainty object which can output the total number of rounds of the game according to pre-defined distribution (by default it uses a geometric distribution).
- **strategies** (*List[egttools.behaviors.AbstractCRDStrategy]*) – A list containing references of AbstractCRDStrategy strategies (or child classes).

See also:

egttools.games.AbstractGame, *egttools.games.NormalFormGame*, *egttools.games.CRDGame*

__new__(***kwargs*)

__str__(*self:* egttools.numerical.games.CRDGameTU) → str

calculate_fitness(*self:* egttools.numerical.games.CRDGameTU, *player_strategy: int*, *pop_size: int*, *population_state: numpy.ndarray[numpy.uint64[m, 1]]*) → float

calculates the fitness of an individual of a given strategy given a population state.It always assumes that the population state does not contain the current individual

calculate_group_achievement(*self:* egttools.numerical.games.CRDGameTU, *population_size:* *int*, *stationary_distribution:* *numpy.ndarray[numpy.float64[m, 1]]*) → float

calculates the group achievement for a given stationary distribution.

calculate_payoffs(*self:* egttools.numerical.games.CRDGameTU) → numpy.ndarray[numpy.float64[m, n]]

updates the internal payoff and coop_level matrices by calculating the payoff of each strategy given any possible strategy pair

calculate_polarization(*self:* egttools.numerical.games.CRDGameTU, *population_size:* *int*, *population_state:* *numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[3, 1]]

calculates the fraction of players that contribute above, below or equal to the fair contribution (E/2)in a give population state.

calculate_polarization_success(*self:* egttools.numerical.games.CRDGameTU, *population_size:* *int*, *population_state:* *numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[3, 1]]

calculates the fraction of players (from successful groups)) that contribute above, below or equal to the fair contribution (E/2)in a give population state.

calculate_population_group_achievement(*self:* egttools.numerical.games.CRDGameTU, *population_size:* *int*, *population_state:* *numpy.ndarray[numpy.uint64[m, 1]]*) → float

calculates the group achievement in the population at a given state.

nb_strategies(*self:* egttools.numerical.games.CRDGameTU) → int

Number of different strategies which are playing the game.

payoff(*self: egttools.numerical.games.CRDGameTU, strategy: int, strategy pair: List[int]*) → float

returns the payoff of a strategy given a group composition.

payoffs(*self:* egttools.numerical.games.CRDGameTU) → numpy.ndarray[numpy.float64[m, n]]

returns the expected payoffs of each strategy vs each possible game state

play(*self:* egttools.numerical.games.CRDGameTU, *arg0: List[int]*, *arg1: List[float]*) → None

save_payoffs(*self:* egttools.numerical.games.CRDGameTU, *arg0:* *str*) → None

Saves the payoff matrix in a txt file.

type(*self:* egttools.numerical.games.CRDGameTU) → str

property endowment

Initial endowment for all players.

property group_size

Size of the group which will play the game.

property min_rounds

Minimum number of rounds of the game.

property nb_states

Number of combinations of 2 strategies that can be matched in the game.

property risk

Probability that all players will lose their remaining endowment if the target si not achieved.

property strategies

A list with pointers to the strategies that are playing the game.

property target

Collective target which needs to be achieved by the group.

2.8.4 egttools.games.InformalRiskGame

class InformalRiskGame(*group_size*, *cost*, *multiplying_factor*, *strategies*)

Bases: *egttools.numerical.games.AbstractGame*

This game has been taken from the model introduced in

`Santos, F. P., Pacheco, J. M., Santos, F. C., & Levin, S. A. (2021). Dynamics of informal risk sharing in collective index insurance. Nature Sustainability. https://doi.org/10.1038/s41893-020-00667-2`

to investigate the dynamics of a collective index insurance with informal risk sharing.

Parameters

- **group_size** (int) –
- **cost** (float) –
- **multiplying_factor** (float) –
- **strategies** (List) –

Methods

calculate_fitness	Calculates the Fitness of an strategy for a given population state.
calculate_payoffs	Estimates the payoffs for each strategy and returns the values in a matrix.
payoff	Returns the payoff of a strategy given a group composition.
play	Updates the vector of payoffs with the payoffs of each player after playing the game.
save_payoffs	Stores the payoff matrix in a txt file.

Attributes

nb_strategies	Number of different strategies playing the game.
payoffs	returns the payoff matrix of the game.
type	returns the type of game.

__init__(*group_size*, *cost*, *multiplying_factor*, *strategies*)

This game has been taken from the model introduced in

`Santos, F. P., Pacheco, J. M., Santos, F. C., & Levin, S. A. (2021). Dynamics of informal risk sharing in collective index insurance. Nature Sustainability. https://doi.org/10.1038/s41893-020-00667-2`

to investigate the dynamics of a collective index insurance with informal risk sharing.

Parameters

- **group_size** (int) –
- **cost** (float) –
- **multiplying_factor** (float) –

- **strategies** (List) –

__new__(***kwargs*)

__str__()

Return type str

calculate_fitness(*player_strategy*, *pop_size*, *population_state*)

Calculates the Fitness of an strategy for a given population state.

The calculation is done by computing the expected payoff over all possible group combinations for the given population state: $ fitness = sum_{states} payoff * P(state) $:type player_strategy: int :param player_strategy: :type player_strategy: index of the strategy :type pop_size: int :param pop_size: (might be eliminated in the future) :type pop_size: size of the population - Only necessary for compatibility with the C++ implementation :type population_state: ndarray :param population_state: :type population_state: vector with the population state (the number of players adopting each strategy)

Returns

Return type The fitness of the population.

calculate_payoffs()

Estimates the payoffs for each strategy and returns the values in a matrix. Each row of the matrix represents a strategy and each column a game state. E.g., in case of a 2 player game, each entry a_ij gives the payoff for strategy i against strategy j. In case of a group game, each entry a_ij gives the payoff of strategy i for game state j, which represents the group composition.

Returns A matrix with the expected payoffs for each strategy given each possible game state.

Return type numpy.ndarray[numpy.float64[m, n]]

payoff(*strategy*, *group_composition*)

Returns the payoff of a strategy given a group composition.

If the group composition does not include the strategy, the payoff should be zero.

Parameters

- **strategy** (*int*) – The index of the strategy used by the player.
- **group_composition** (*List[int]*) – List with the group composition. The structure of this list depends on the particular implementation of this abstract method.

Returns The payoff value.

Return type float

play(*group_composiiton*, *game_payoffs*)

Updates the vector of payoffs with the payoffs of each player after playing the game.

This method will run the game using the players and player types defined in :param group_composition, and will update the vector :param game_payoffs with the resulting payoff of each player.

Parameters

- **group_composition** (*List[int]*) – A list with counts of the number of players of each strategy in the group.
- **game_payoffs** (*List[float]*) – A list used as container for the payoffs of each player

Return type None

save_payoffs(*file_name*)

Stores the payoff matrix in a txt file.

Parameters **file_name** (*str*) – Name of the file in which the data will be stored.

Return type None

property nb_strategies: int

Number of different strategies playing the game.

Return type int

property payoffs: numpy.ndarray

returns the payoff matrix of the game.

Return type ndarray

property type: str

returns the type of game.

Return type str

2.8.5 egttools.games.NormalFormGame

class NormalFormGame(**args*, ***kwargs*)

Bases: *egttools.numerical.games.AbstractGame*

Overloaded function.

1. __init__(self: egttools.numerical.games.NormalFormGame, nb_rounds: int, payoff_matrix: numpy.ndarray[numpy.float64[m, n], flags.c_contiguous]) -> None

   Normal Form Game. This constructor assumes that there are only two possible strategies and two possible actions.

   This method will run the game using the players and player types defined in :param group_composition, and will update the vector :param game_payoffs with the resulting payoff of each player.

   nb_rounds [int] Number of rounds of the game.

   payoff_matrix [numpy.ndarray[numpy.float64[m, m]]] A payoff matrix of shape (nb_actions, nb_actions).

   egttools.games.AbstractGame

2. __init__(self: egttools.numerical.games.NormalFormGame, nb_rounds: int, payoff_matrix: numpy.ndarray[numpy.float64[m, n], flags.c_contiguous], strategies: list) -> None

   Normal Form Game. This constructor allows you to define any number of strategies by passing a list of pointers to them. All strategies must by of type AbstractNFGStrategy *.

   nb_rounds [int] Number of rounds of the game.

   payoff_matrix [numpy.ndarray[float]] A payoff matrix of shape (nb_actions, nb_actions).

   strategies [List[egttools.behaviors.AbstractNFGStrategy]] A list containing references of AbstractNFGStrategy strategies (or child classes).

   egttools.games.AbstractGame

Methods

calculate_cooperation_rate	calculates the rate/level of cooperation in the population at a given state.
calculate_fitness	calculates the fitness of an individual of a given strategy given a population state.It always assumes that the population state does not contain the current individual
calculate_payoffs	updates the internal payoff and coop_level matrices by calculating the payoff of each strategy given any possible strategy pair
expected_payoffs	returns the expected payoffs of each strategy vs another
nb_strategies	Number of different strategies which are playing the game.
payoff	returns the payoff of a strategy given a strategy pair.
payoffs	
play	
save_payoffs	Saves the payoff matrix in a txt file.
type	

Attributes

nb_rounds	Number of rounds of the game.
nb_states	Number of combinations of 2 strategies that can be matched in the game.
strategies	A list with pointers to the strategies that are playing the game.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: egttools.numerical.games.NormalFormGame, nb_rounds: int, payoff_matrix: numpy.ndarray[numpy.float64[m, n], flags.c_contiguous]) -> None

   Normal Form Game. This constructor assumes that there are only two possible strategies and two possible actions.

   This method will run the game using the players and player types defined in :param group_composition, and will update the vector :param game_payoffs with the resulting payoff of each player.

   nb_rounds [int] Number of rounds of the game.

   payoff_matrix [numpy.ndarray[numpy.float64[m, m]]] A payoff matrix of shape (nb_actions, nb_actions).

   egttools.games.AbstractGame

2. __init__(self: egttools.numerical.games.NormalFormGame, nb_rounds: int, payoff_matrix: numpy.ndarray[numpy.float64[m, n], flags.c_contiguous], strategies: list) -> None

Normal Form Game. This constructor allows you to define any number of strategies by passing a list of pointers to them. All strategies must by of type AbstractNFGStrategy *.

nb_rounds [int] Number of rounds of the game.

payoff_matrix [numpy.ndarray[float]] A payoff matrix of shape (nb_actions, nb_actions).

strategies [List[egttools.behaviors.AbstractNFGStrategy]] A list containing references of AbstractNFGStrategy strategies (or child classes).

egttools.games.AbstractGame

__new__(***kwargs*)

__str__(*self:* egttools.numerical.games.NormalFormGame) → str

calculate_cooperation_rate(*self:* egttools.numerical.games.NormalFormGame, *population_size: int*, *population_state: numpy.ndarray[numpy.uint64[m, 1]]*) → float

calculates the rate/level of cooperation in the population at a given state.

calculate_fitness(*self:* egttools.numerical.games.NormalFormGame, *player_strategy: int*, *pop_size: int*, *population_state: numpy.ndarray[numpy.uint64[m, 1]]*) → float

calculates the fitness of an individual of a given strategy given a population state.It always assumes that the population state does not contain the current individual

calculate_payoffs(*self:* egttools.numerical.games.NormalFormGame) → numpy.ndarray[numpy.float64[m, n]]

updates the internal payoff and coop_level matrices by calculating the payoff of each strategy given any possible strategy pair

expected_payoffs(*self:* egttools.numerical.games.NormalFormGame) → numpy.ndarray[numpy.float64[m, n]]

returns the expected payoffs of each strategy vs another

nb_strategies(*self:* egttools.numerical.games.NormalFormGame) → int

Number of different strategies which are playing the game.

payoff(*self: egttools.numerical.games.NormalFormGame, strategy: int, strategy pair: List[int]*) → float

returns the payoff of a strategy given a strategy pair.

payoffs(*self:* egttools.numerical.games.NormalFormGame) → numpy.ndarray[numpy.float64[m, n]]

play(*self:* egttools.numerical.games.NormalFormGame, *arg0: List[int]*, *arg1: List[float]*) → None

save_payoffs(*self:* egttools.numerical.games.NormalFormGame, *arg0: str*) → None

Saves the payoff matrix in a txt file.

type(*self:* egttools.numerical.games.NormalFormGame) → str

property nb_rounds

Number of rounds of the game.

property nb_states

Number of combinations of 2 strategies that can be matched in the game.

property strategies

A list with pointers to the strategies that are playing the game.

2.8.6 egttools.games.PGG

class PGG(*group_size*, *cost*, *multiplying_factor*, *strategies*)

Bases: `egttools.numerical.games.AbstractGame`

Methods

`calculate_fitness`	Calculates the Fitness of an strategy for a given population state.
`calculate_payoffs`	Estimates the payoffs for each strategy and returns the values in a matrix.
`nb_strategies`	Number of different strategies playing the game.
`payoff`	Returns the payoff of a strategy given a group composition.
`payoffs`	returns the payoff matrix of the game.
`play`	Updates the vector of payoffs with the payoffs of each player after playing the game.
`save_payoffs`	Stores the payoff matrix in a txt file.
`type`	returns the type of game.

__init__(*group_size*, *cost*, *multiplying_factor*, *strategies*)

__new__(***kwargs*)

__str__()

Return type `str`

calculate_fitness(*player_strategy*, *pop_size*, *population_state*)

Calculates the Fitness of an strategy for a given population state.

The calculation is done by computing the expected payoff over all possible group combinations for the given population state: $ fitness = sum_{states} payoff * P(state) $:type player_strategy: `int` :param player_strategy: :type player_strategy: index of the strategy :type pop_size: `int` :param pop_size: (might be eliminated in the future) :type pop_size: size of the population - Only necessary for compatibility with the C++ implementation :type population_state: `ndarray` :param population_state: :type population_state: vector with the population state (the number of players adopting each strategy)

Returns

Return type The fitness of the population.

calculate_payoffs()

Estimates the payoffs for each strategy and returns the values in a matrix. Each row of the matrix represents a strategy and each column a game state. E.g., in case of a 2 player game, each entry a_ij gives the payoff for strategy i against strategy j. In case of a group game, each entry a_ij gives the payoff of strategy i for game state j, which represents the group composition.

Returns A matrix with the expected payoffs for each strategy given each possible game state.

Return type numpy.ndarray[numpy.float64[m, n]]

nb_strategies()

Number of different strategies playing the game.

Return type `int`

payoff(*strategy*, *group_composition*)

Returns the payoff of a strategy given a group composition.

If the group composition does not include the strategy, the payoff should be zero.

Parameters

- **strategy** (*int*) – The index of the strategy used by the player.
- **group_composition** (*List[int]*) – List with the group composition. The structure of this list depends on the particular implementation of this abstract method.

Returns The payoff value.

Return type float

payoffs()

returns the payoff matrix of the game.

Return type ndarray

play(*group_composiiton*, *game_payoffs*)

Updates the vector of payoffs with the payoffs of each player after playing the game.

This method will run the game using the players and player types defined in :param group_composition, and will update the vector :param game_payoffs with the resulting payoff of each player.

Parameters

- **group_composition** (*List[int]*) – A list with counts of the number of players of each strategy in the group.
- **game_payoffs** (*List[float]*) – A list used as container for the payoffs of each player

Return type None

save_payoffs(*file_name*)

Stores the payoff matrix in a txt file.

Parameters **file_name** (*str*) – Name of the file in which the data will be stored.

Return type None

type()

returns the type of game.

Return type str

---

egttools.games.informal_risk

---

egttools.games.pgg

2.8.7 egttools.games.informal_risk

Functions

calculate_nb_states	Calculates the number of states (combinations) of the members of a group in a subgroup.
calculate_state	Overloaded function.
sample_simplex	Transforms a state index into a vector.

egttools.games.informal_risk.calculate_nb_states

calculate_nb_states(*group_size: int*, *nb_strategies: int*) → int

Calculates the number of states (combinations) of the members of a group in a subgroup. It can be used to calculate the maximum number of states in a discrete simplex.

The implementation of this method follows the stars and bars algorithm (see Wikipedia).

Parameters

- **group_size** (*int*) – Size of the group (maximum number of players/elements that can adopt each possible strategy).
- **nb_strategies** (*int*) – number of strategies that can be assigned to players.

Returns Number of states (possible combinations of strategies and players).

Return type int

See also:

egttools.numerical.calculate_state, *egttools.numerical.sample_simplex*

egttools.games.informal_risk.calculate_state

calculate_state(**args*, ***kwargs*)

Overloaded function.

1. calculate_state(group_size: int, group_composition: List[int]) -> int

   This function converts a vector containing counts into an index.

   This method was copied from @Svalorzen.

   group_size [int] Maximum bin size (it can also be the population size).

   group_composition [List[int]] The vector to convert from simplex coordinates to index.

   int The unique index in [0, egttools.calculate_nb_states(group_size, len(group_composition)) representing the n-dimensional simplex.

   egttools.sample_simplex, egttools.calculate_nb_states

2. calculate_state(group_size: int, group_composition: numpy.ndarray[numpy.uint64[m, 1]]) -> int

   This function converts a vector containing counts into an index.

   This method was copied from @Svalorzen.

   group_size [int] Maximum bin size (it can also be the population size).

group_composition [numpy.ndarray[numpy.int64[m, 1]]] The vector to convert from simplex coordinates to index.

int The unique index in [0, egttools.calculate_nb_states(group_size, len(group_composition)) representing the n-dimensional simplex.

egttools.sample_simplex, egttools.calculate_nb_states

egttools.games.informal_risk.sample_simplex

sample_simplex(*index: int*, *pop_size: int*, *nb_strategies: int*) → numpy.ndarray[numpy.uint64[m, 1]]

Transforms a state index into a vector.

Parameters

- **index** (*int*) – State index.
- **pop_size** (*int*) – Size of the population.
- **nb_strategies** (*int*) – Number of strategies.

Returns Vector with the sampled state.

Return type numpy.ndarray[numpy.int64[m, 1]]

See also:

egttools.numerical.calculate_state, *egttools.numerical.calculate_nb_states*

Classes

AbstractGame	
InformalRiskGame	This game has been taken from the model introduced in

egttools.games.informal_risk.AbstractGame

class AbstractGame(*self:* egttools.numerical.games.AbstractGame) → None

Bases: `pybind11_builtins.pybind11_object`

Methods

calculate_fitness	Estimates the fitness for a player_type in the population with state :param strategies.
calculate_payoffs	Estimates the payoffs for each strategy and returns the values in a matrix.
nb_strategies	Number of different strategies playing the game.
payoff	Returns the payoff of a strategy given a group composition.
payoffs	returns the payoff matrix of the game.
play	Updates the vector of payoffs with the payoffs of each player after playing the game.

continues on next page

Table 44 – continued from previous page

save_payoffs	Stores the payoff matrix in a txt file.
type	returns the type of game.

__init__(*self:* egttools.numerical.games.AbstractGame) → None

__new__(***kwargs*)

__str__(*self:* egttools.numerical.games.AbstractGame) → str

calculate_fitness(*self:* egttools.numerical.games.AbstractGame, *player_type: int*, *pop_size: int*, *strategies: numpy.ndarray[numpy.uint64[m, 1]]*) → float

Estimates the fitness for a player_type in the population with state :param strategies.

This function assumes that the player with strategy player_type is not included in the vector of strategy counts strategies.

Parameters

- **player_type** (*int*) – The index of the strategy used by the player.
- **pop_size** (*int*) – The size of the population.
- **strategies** (*numpy.ndarray[numpy.uint64[m, 1]]*) – A vector of counts of each strategy. The current state of the population.

Returns The fitness of the strategy in the population state given by strategies.

Return type float

calculate_payoffs(*self:* egttools.numerical.games.AbstractGame) → numpy.ndarray[numpy.float64[m, n]]

Estimates the payoffs for each strategy and returns the values in a matrix. Each row of the matrix represents a strategy and each column a game state. E.g., in case of a 2 player game, each entry a_ij gives the payoff for strategy i against strategy j. In case of a group game, each entry a_ij gives the payoff of strategy i for game state j, which represents the group composition.

Returns A matrix with the expected payoffs for each strategy given each possible game state.

Return type numpy.ndarray[numpy.float64[m, n]]

nb_strategies(*self:* egttools.numerical.games.AbstractGame) → int

Number of different strategies playing the game.

payoff(*self:* egttools.numerical.games.AbstractGame, *strategy: int*, *group_composition: List[int]*) → float

Returns the payoff of a strategy given a group composition.

If the group composition does not include the strategy, the payoff should be zero.

Parameters

- **strategy** (*int*) – The index of the strategy used by the player.
- **group_composition** (*List[int]*) – List with the group composition. The structure of this list depends on the particular implementation of this abstract method.

Returns The payoff value.

Return type float

payoffs(*self:* egttools.numerical.games.AbstractGame) → numpy.ndarray[numpy.float64[m, n]]

returns the payoff matrix of the game.

play(*self:* egttools.numerical.games.AbstractGame, *group_composition: List[int]*, *game_payoffs: List[float]*) → None

Updates the vector of payoffs with the payoffs of each player after playing the game.

This method will run the game using the players and player types defined in :param group_composition, and will update the vector :param game_payoffs with the resulting payoff of each player.

Parameters

- **group_composition** (*List[int]*) – A list with counts of the number of players of each strategy in the group.
- **game_payoffs** (*List[float]*) – A list used as container for the payoffs of each player

save_payoffs(*self:* egttools.numerical.games.AbstractGame, *file_name: str*) → None
Stores the payoff matrix in a txt file.

Parameters **file_name** (*str*) – Name of the file in which the data will be stored.

type(*self:* egttools.numerical.games.AbstractGame) → str
returns the type of game.

egttools.games.informal_risk.InformalRiskGame

class InformalRiskGame(*group_size*, *cost*, *multiplying_factor*, *strategies*)
Bases: *egttools.numerical.games.AbstractGame*

This game has been taken from the model introduced in

`Santos, F. P., Pacheco, J. M., Santos, F. C., & Levin, S. A. (2021). Dynamics of informal risk sharing in collective index insurance. Nature Sustainability. https://doi.org/10.1038/s41893-020-00667-2`

to investigate the dynamics of a collective index insurance with informal risk sharing.

Parameters

- **group_size** (int) –
- **cost** (float) –
- **multiplying_factor** (float) –
- **strategies** (List) –

Methods

calculate_fitness	Calculates the Fitness of an strategy for a given population state.
calculate_payoffs	Estimates the payoffs for each strategy and returns the values in a matrix.
payoff	Returns the payoff of a strategy given a group composition.
play	Updates the vector of payoffs with the payoffs of each player after playing the game.
save_payoffs	Stores the payoff matrix in a txt file.

Attributes

nb_strategies	Number of different strategies playing the game.
payoffs	returns the payoff matrix of the game.
type	returns the type of game.

__init__(*group_size*, *cost*, *multiplying_factor*, *strategies*)

This game has been taken from the model introduced in

`Santos, F. P., Pacheco, J. M., Santos, F. C., & Levin, S. A. (2021). Dynamics of informal risk sharing in collective index insurance. Nature Sustainability. https://doi.org/10.1038/s41893-020-00667-2`

to investigate the dynamics of a collective index insurance with informal risk sharing.

Parameters

- **group_size** (int) –
- **cost** (float) –
- **multiplying_factor** (float) –
- **strategies** (List) –

__new__(**kwargs)

__str__()

Return type str

calculate_fitness(*player_strategy*, *pop_size*, *population_state*)

Calculates the Fitness of an strategy for a given population state.

The calculation is done by computing the expected payoff over all possible group combinations for the given population state: $ fitness = sum_{states} payoff * P(state) $:type player_strategy: int :param player_strategy: :type player_strategy: index of the strategy :type pop_size: int :param pop_size: (might be eliminated in the future) :type pop_size: size of the population - Only necessary for compatibility with the C++ implementation :type population_state: ndarray :param population_state: :type population_state: vector with the population state (the number of players adopting each strategy)

Returns

Return type The fitness of the population.

calculate_payoffs()

Estimates the payoffs for each strategy and returns the values in a matrix. Each row of the matrix represents a strategy and each column a game state. E.g., in case of a 2 player game, each entry a_ij gives the payoff for strategy i against strategy j. In case of a group game, each entry a_ij gives the payoff of strategy i for game state j, which represents the group composition.

Returns A matrix with the expected payoffs for each strategy given each possible game state.

Return type numpy.ndarray[numpy.float64[m, n]]

payoff(*strategy*, *group_composition*)

Returns the payoff of a strategy given a group composition.

If the group composition does not include the strategy, the payoff should be zero.

Parameters

- **strategy** (*int*) – The index of the strategy used by the player.
- **group_composition** (*List[int]*) – List with the group composition. The structure of this list depends on the particular implementation of this abstract method.

Returns The payoff value.

Return type float

play(*group_composiiton*, *game_payoffs*)

Updates the vector of payoffs with the payoffs of each player after playing the game.

This method will run the game using the players and player types defined in :param group_composition, and will update the vector :param game_payoffs with the resulting payoff of each player.

Parameters

- **group_composition** (*List[int]*) – A list with counts of the number of players of each strategy in the group.
- **game_payoffs** (*List[float]*) – A list used as container for the payoffs of each player

Return type None

save_payoffs(*file_name*)

Stores the payoff matrix in a txt file.

Parameters **file_name** (*str*) – Name of the file in which the data will be stored.

Return type None

property nb_strategies: int

Number of different strategies playing the game.

Return type int

property payoffs: numpy.ndarray

returns the payoff matrix of the game.

Return type ndarray

property type: str

returns the type of game.

Return type str

2.8.8 egttools.games.pgg

Functions

calculate_nb_states	Calculates the number of states (combinations) of the members of a group in a subgroup.
calculate_state	Overloaded function.
sample_simplex	Transforms a state index into a vector.

egttools.games.pgg.calculate_nb_states

calculate_nb_states(*group_size: int*, *nb_strategies: int*) → int

Calculates the number of states (combinations) of the members of a group in a subgroup. It can be used to calculate the maximum number of states in a discrete simplex.

The implementation of this method follows the stars and bars algorithm (see Wikipedia).

Parameters

- **group_size** (*int*) – Size of the group (maximum number of players/elements that can adopt each possible strategy).
- **nb_strategies** (*int*) – number of strategies that can be assigned to players.

Returns Number of states (possible combinations of strategies and players).

Return type int

See also:

egttools.numerical.calculate_state, *egttools.numerical.sample_simplex*

egttools.games.pgg.calculate_state

calculate_state(**args*, ***kwargs*)

Overloaded function.

1. calculate_state(group_size: int, group_composition: List[int]) -> int

   This function converts a vector containing counts into an index.

   This method was copied from @Svalorzen.

   group_size [int] Maximum bin size (it can also be the population size).

   group_composition [List[int]] The vector to convert from simplex coordinates to index.

   int The unique index in [0, egttools.calculate_nb_states(group_size, len(group_composition)) representing the n-dimensional simplex.

   egttools.sample_simplex, egttools.calculate_nb_states

2. calculate_state(group_size: int, group_composition: numpy.ndarray[numpy.uint64[m, 1]]) -> int

   This function converts a vector containing counts into an index.

   This method was copied from @Svalorzen.

   group_size [int] Maximum bin size (it can also be the population size).

   group_composition [numpy.ndarray[numpy.int64[m, 1]]] The vector to convert from simplex coordinates to index.

   int The unique index in [0, egttools.calculate_nb_states(group_size, len(group_composition)) representing the n-dimensional simplex.

   egttools.sample_simplex, egttools.calculate_nb_states

egttools.games.pgg.sample_simplex

sample_simplex(*index: int*, *pop_size: int*, *nb_strategies: int*) → numpy.ndarray[numpy.uint64[m, 1]]

Transforms a state index into a vector.

Parameters

- **index** (*int*) – State index.
- **pop_size** (*int*) – Size of the population.
- **nb_strategies** (*int*) – Number of strategies.

Returns Vector with the sampled state.

Return type numpy.ndarray[numpy.int64[m, 1]]

See also:

egttools.numerical.calculate_state, *egttools.numerical.calculate_nb_states*

Classes

AbstractGame
PGG

egttools.games.pgg.AbstractGame

class AbstractGame(*self:* egttools.numerical.games.AbstractGame) → None

Bases: `pybind11_builtins.pybind11_object`

Methods

calculate_fitness	Estimates the fitness for a player_type in the population with state :param strategies.
calculate_payoffs	Estimates the payoffs for each strategy and returns the values in a matrix.
nb_strategies	Number of different strategies playing the game.
payoff	Returns the payoff of a strategy given a group composition.
payoffs	returns the payoff matrix of the game.
play	Updates the vector of payoffs with the payoffs of each player after playing the game.
save_payoffs	Stores the payoff matrix in a txt file.
type	returns the type of game.

__init__(*self:* egttools.numerical.games.AbstractGame) → None

__new__(***kwargs*)

__str__(*self:* egttools.numerical.games.AbstractGame) → str

calculate_fitness(*self:* egttools.numerical.games.AbstractGame, *player_type: int*, *pop_size: int*, *strategies: numpy.ndarray[numpy.uint64[m, 1]]*) → float

Estimates the fitness for a player_type in the population with state :param strategies.

This function assumes that the player with strategy player_type is not included in the vector of strategy counts strategies.

Parameters

- **player_type** (*int*) – The index of the strategy used by the player.
- **pop_size** (*int*) – The size of the population.
- **strategies** (*numpy.ndarray[numpy.uint64[m, 1]]*) – A vector of counts of each strategy. The current state of the population.

Returns The fitness of the strategy in the population state given by strategies.

Return type float

calculate_payoffs(*self:* egttools.numerical.games.AbstractGame) → numpy.ndarray[numpy.float64[m, n]]

Estimates the payoffs for each strategy and returns the values in a matrix. Each row of the matrix represents a strategy and each column a game state. E.g., in case of a 2 player game, each entry a_ij gives the payoff for strategy i against strategy j. In case of a group game, each entry a_ij gives the payoff of strategy i for game state j, which represents the group composition.

Returns A matrix with the expected payoffs for each strategy given each possible game state.

Return type numpy.ndarray[numpy.float64[m, n]]

nb_strategies(*self:* egttools.numerical.games.AbstractGame) → int

Number of different strategies playing the game.

payoff(*self:* egttools.numerical.games.AbstractGame, *strategy: int*, *group_composition: List[int]*) → float

Returns the payoff of a strategy given a group composition.

If the group composition does not include the strategy, the payoff should be zero.

Parameters

- **strategy** (*int*) – The index of the strategy used by the player.
- **group_composition** (*List[int]*) – List with the group composition. The structure of this list depends on the particular implementation of this abstract method.

Returns The payoff value.

Return type float

payoffs(*self:* egttools.numerical.games.AbstractGame) → numpy.ndarray[numpy.float64[m, n]]

returns the payoff matrix of the game.

play(*self:* egttools.numerical.games.AbstractGame, *group_composition: List[int]*, *game_payoffs: List[float]*) → None

Updates the vector of payoffs with the payoffs of each player after playing the game.

This method will run the game using the players and player types defined in :param group_composition, and will update the vector :param game_payoffs with the resulting payoff of each player.

Parameters

- **group_composition** (*List[int]*) – A list with counts of the number of players of each strategy in the group.
- **game_payoffs** (*List[float]*) – A list used as container for the payoffs of each player

save_payoffs(*self:* egttools.numerical.games.AbstractGame, *file_name: str*) → None
Stores the payoff matrix in a txt file.

Parameters **file_name** (*str*) – Name of the file in which the data will be stored.

type(*self:* egttools.numerical.games.AbstractGame) → str
returns the type of game.

egttools.games.pgg.PGG

class PGG(*group_size*, *cost*, *multiplying_factor*, *strategies*)
Bases: *egttools.numerical.games.AbstractGame*

Methods

calculate_fitness	Calculates the Fitness of an strategy for a given population state.
calculate_payoffs	Estimates the payoffs for each strategy and returns the values in a matrix.
nb_strategies	Number of different strategies playing the game.
payoff	Returns the payoff of a strategy given a group composition.
payoffs	returns the payoff matrix of the game.
play	Updates the vector of payoffs with the payoffs of each player after playing the game.
save_payoffs	Stores the payoff matrix in a txt file.
type	returns the type of game.

__init__(*group_size*, *cost*, *multiplying_factor*, *strategies*)

__new__(***kwargs*)

__str__()

Return type str

calculate_fitness(*player_strategy*, *pop_size*, *population_state*)
Calculates the Fitness of an strategy for a given population state.

The calculation is done by computing the expected payoff over all possible group combinations for the given population state: $ fitness = sum_{states} payoff * P(state) $:type player_strategy: int :param player_strategy: :type player_strategy: index of the strategy :type pop_size: int :param pop_size: (might be eliminated in the future) :type pop_size: size of the population - Only necessary for compatibility with the C++ implementation :type population_state: ndarray :param population_state: :type population_state: vector with the population state (the number of players adopting each strategy)

Returns

Return type The fitness of the population.

calculate_payoffs()
Estimates the payoffs for each strategy and returns the values in a matrix. Each row of the matrix represents a strategy and each column a game state. E.g., in case of a 2 player game, each entry a_ij gives the payoff for strategy i against strategy j. In case of a group game, each entry a_ij gives the payoff of strategy i for game state j, which represents the group composition.

Returns A matrix with the expected payoffs for each strategy given each possible game state.

Return type numpy.ndarray[numpy.float64[m, n]]

nb_strategies()

Number of different strategies playing the game.

Return type int

payoff(*strategy*, *group_composition*)

Returns the payoff of a strategy given a group composition.

If the group composition does not include the strategy, the payoff should be zero.

Parameters

- **strategy** (*int*) – The index of the strategy used by the player.
- **group_composition** (*List[int]*) – List with the group composition. The structure of this list depends on the particular implementation of this abstract method.

Returns The payoff value.

Return type float

payoffs()

returns the payoff matrix of the game.

Return type ndarray

play(*group_composiiton*, *game_payoffs*)

Updates the vector of payoffs with the payoffs of each player after playing the game.

This method will run the game using the players and player types defined in :param group_composition, and will update the vector :param game_payoffs with the resulting payoff of each player.

Parameters

- **group_composition** (*List[int]*) – A list with counts of the number of players of each strategy in the group.
- **game_payoffs** (*List[float]*) – A list used as container for the payoffs of each player

Return type None

save_payoffs(*file_name*)

Stores the payoff matrix in a txt file.

Parameters **file_name** (*str*) – Name of the file in which the data will be stored.

Return type None

type()

returns the type of game.

Return type str

2.9 egttools.numerical

The `numerical` module contains optimized functions and classes to simulate evolutionary dynamics in large populations. This module is written in C++.

Functions

calculate_nb_states	Calculates the number of states (combinations) of the members of a group in a subgroup.
calculate_state	Overloaded function.
calculate_strategies_distribution	Calculates the average frequency of each strategy available in the population given the stationary distribution.
call_get_action	Returns a string with a tuple of actions
sample_simplex	Transforms a state index into a vector.
sample_simplex_directly	Samples an N-dimensional point directly from the simplex.
sample_unit_simplex	Samples uniformly at random the unit simplex with nb_strategies dimensionse.

2.9.1 egttools.numerical.calculate_nb_states

calculate_nb_states(*group_size: int*, *nb_strategies: int*) → int

Calculates the number of states (combinations) of the members of a group in a subgroup. It can be used to calculate the maximum number of states in a discrete simplex.

The implementation of this method follows the stars and bars algorithm (see Wikipedia).

Parameters

- **group_size** (*int*) – Size of the group (maximum number of players/elements that can adopt each possible strategy).
- **nb_strategies** (*int*) – number of strategies that can be assigned to players.

Returns Number of states (possible combinations of strategies and players).

Return type int

See also:

egttools.numerical.calculate_state, *egttools.numerical.sample_simplex*

2.9.2 egttools.numerical.calculate_state

calculate_state(**args*, ***kwargs*)

Overloaded function.

1. calculate_state(group_size: int, group_composition: List[int]) -> int

   This function converts a vector containing counts into an index.

   This method was copied from @Svalorzen.

   group_size [int] Maximum bin size (it can also be the population size).

   group_composition [List[int]] The vector to convert from simplex coordinates to index.

int The unique index in [0, egttools.calculate_nb_states(group_size, len(group_composition)) representing the n-dimensional simplex.

egttools.sample_simplex, egttools.calculate_nb_states

2. calculate_state(group_size: int, group_composition: numpy.ndarray[numpy.uint64[m, 1]]) -> int

This function converts a vector containing counts into an index.

This method was copied from @Svalorzen.

group_size [int] Maximum bin size (it can also be the population size).

group_composition [numpy.ndarray[numpy.int64[m, 1]]] The vector to convert from simplex coordinates to index.

int The unique index in [0, egttools.calculate_nb_states(group_size, len(group_composition)) representing the n-dimensional simplex.

egttools.sample_simplex, egttools.calculate_nb_states

2.9.3 egttools.numerical.calculate_strategies_distribution

calculate_strategies_distribution(*pop_size: int, nb_strategies: int, stationary_distribution: scipy.sparse.csr_matrix[numpy.float64]*) → numpy.ndarray[numpy.float64[m, 1]]

Calculates the average frequency of each strategy available in the population given the stationary distribution.

Parameters

- **pop_size** (*int*) – Size of the population.
- **nb_strategies** (*int*) – Number of strategies that can be assigned to players.
- **stationary_distribution** (*scipy.sparse.csr_matrix*) – A sparse matrix which contains the stationary distribution (the frequency with which the evolutionary system visits each stationary state).

Returns Average frequency of each strategy in the stationary evolutionary system.

Return type numpy.ndarray[numpy.float64[m, 1]]

See also:

egttools.numerical.calculate_state, *egttools.numerical.calculate_nb_states*, *egttools.numerical.sample_simplex*, *egttools.numerical.PairwiseMoran.stationary_distribution_sparse*

2.9.4 egttools.numerical.call_get_action

call_get_action(*strategies: list, time_step: int, prev_action: int*) → str

Returns a string with a tuple of actions

2.9.5 egttools.numerical.sample_simplex

sample_simplex(*index: int*, *pop_size: int*, *nb_strategies: int*) → numpy.ndarray[numpy.uint64[m, 1]]

Transforms a state index into a vector.

Parameters

- **index** (*int*) – State index.
- **pop_size** (*int*) – Size of the population.
- **nb_strategies** (*int*) – Number of strategies.

Returns Vector with the sampled state.

Return type numpy.ndarray[numpy.int64[m, 1]]

See also:

egttools.numerical.calculate_state, *egttools.numerical.calculate_nb_states*

2.9.6 egttools.numerical.sample_simplex_directly

sample_simplex_directly(*nb_strategies: int*, *pop_size: int*) → numpy.ndarray[numpy.int64[m, 1]]

Samples an N-dimensional point directly from the simplex. N is the number of strategies.

Parameters

- **nb_strategies** (*int*) – Number of strategies.
- **pop_size** (*int*) – Size of the population.

Returns Vector with the sampled state.

Return type numpy.ndarray[numpy.int64[m, 1]]

See also:

egttools.numerical.calculate_state, *egttools.numerical.calculate_nb_states*, *egttools.numerical.sample_simplex*

2.9.7 egttools.numerical.sample_unit_simplex

sample_unit_simplex(*nb_strategies: int*) → numpy.ndarray[numpy.float64[m, 1]]

Samples uniformly at random the unit simplex with nb_strategies dimensionse.

Parameters **nb_strategies** (*int*) – Number of strategies.

Returns Vector with the sampled state.

Return type numpy.ndarray[numpy.int64[m, 1]]

See also:

egttools.numerical.calculate_state, *egttools.numerical.calculate_nb_states*, *egttools.numerical.sample_simplex*

Classes

PairwiseMoran	Runs a moran process with pairwise comparison and calculates fitness according to game
Random	Random seed generator.

2.9.8 egttools.numerical.PairwiseMoran

class PairwiseMoran(*self:* egttools.numerical.PairwiseMoran, *pop_size:* *int*, *game:* egttools.numerical.games.AbstractGame, *cache_size:* *int*) → None

Bases: `pybind11_builtins.pybind11_object`

Runs a moran process with pairwise comparison and calculates fitness according to game

Methods

estimate_strategy_distribution	Estimates the distribution of strategies in the population given the current game.
evolve	evolves the strategies for a maximum of nb_generations
fixation_probability	Estimates the fixation probability of an strategy in the population.
run	Overloaded function.
stationary_distribution	Estimates the stationary distribution of the population of strategies given the game.
stationary_distribution_sparse	Estimates the stationary distribution of the population of strategies given the game.

Attributes

cache_size	Maximum memory which can be used to cache the fitness calculations.
nb_strategies	Number of strategies in the population.
payoffs	Payoff matrix containing the payoff of each strategy (row) for each game state (column)
pop_size	Size of the population.

__init__(*self:* egttools.numerical.PairwiseMoran, *pop_size:* *int*, *game:* egttools.numerical.games.AbstractGame, *cache_size:* *int*) → None

Runs a moran process with pairwise comparison and calculates fitness according to game

__new__(***kwargs*)

estimate_strategy_distribution(*self:* egttools.numerical.PairwiseMoran, *nb_runs:* *int*, *nb_generations:* *int*, *transitory:* *int*, *beta:* *float*, *mu:* *float*) → numpy.ndarray[numpy.float64[m, 1]]

Estimates the distribution of strategies in the population given the current game.

This method directly estimates how frequent each strategy is in the population, without calculating the stationary distribution as an intermediary step. You should use this method when the number of states of

the system is bigger than MAX_LONG_INT, since it would not be possible to index the states in this case, and stationaryDistribution and estimate_stationary_distribution_sparse would run into an overflow error.

Parameters

- **nb_runs** (*int*) – Number of independent simulations to perform. The final result will be an average over all the runs.
- **nb_generations** (*int*) – Total number of generations.
- **transitory** (*int*) – Transitory period. These generations will be excluded from the final average. Thus, only the last nb_generations - transitory generations will be taken into account. This is important, since in order to obtain a correct average at the steady state, we need to skip the transitory period.
- **beta** (*float*) – Intensity of selection. This parameter determines how important the difference in payoff between players is for the probability of imitation. If beta is small, the system will mostly undergo random drift between strategies. If beta is high, a slight difference in payoff will make a strategy disapear.
- **mu** (*float*) – Probability of mutation. This parameter defines how likely it is for a mutation event to occur at a given generation

Returns The average frequency of each strategy in the population.

Return type numpy.ndarray[numpy.float64[m, 1]]

See also:

egttools.numerical.PairwiseMoran.stationary_distribution, *egttools.numerical.PairwiseMoran.stationary_distribution_sparse*

evolve(*self:* egttools.numerical.PairwiseMoran, *nb_generations: int*, *beta: float*, *mu: float*, *init_state: numpy.ndarray[numpy.uint64[m, 1]]*) → numpy.ndarray[numpy.uint64[m, 1]]

evolves the strategies for a maximum of nb_generations

fixation_probability(*self:* egttools.numerical.PairwiseMoran, *mutant: int*, *resident: int*, *nb_runs: int*, *nb_generations: int*, *beta: float*) → float

Estimates the fixation probability of an strategy in the population.

run(**args*, ***kwargs*)

Overloaded function.

1. run(self: egttools.numerical.PairwiseMoran, nb_generations: int, beta: float, init_state: numpy.ndarray[numpy.uint64[m, 1]]) -> numpy.ndarray[numpy.uint64[m, n]]

runs the moran process with social imitation and returns a matrix with all the states the system went through

2. run(self: egttools.numerical.PairwiseMoran, nb_generations: int, beta: float, mu: float, init_state: numpy.ndarray[numpy.uint64[m, 1]]) -> numpy.ndarray[numpy.uint64[m, n]]

runs the moran process with social imitation and returns a matrix with all the states the system went through

stationary_distribution(*self:* egttools.numerical.PairwiseMoran, *nb_runs: int*, *nb_generations: int*, *transitory: int*, *beta: float*, *mu: float*) → numpy.ndarray[numpy.float64[m, 1]]

Estimates the stationary distribution of the population of strategies given the game.

stationary_distribution_sparse(*self:* egttools.numerical.PairwiseMoran, *nb_runs: int*, *nb_generations: int*, *transitory: int*, *beta: float*, *mu: float*) → scipy.sparse.csr_matrix[numpy.float64]

Estimates the stationary distribution of the population of strategies given the game.

property cache_size

Maximum memory which can be used to cache the fitness calculations.

property nb_strategies
Number of strategies in the population.

property payoffs
Payoff matrix containing the payoff of each strategy (row) for each game state (column)

property pop_size
Size of the population.

2.9.9 egttools.numerical.Random

class Random
Bases: `pybind11_builtins.pybind11_object`

Random seed generator.

Methods

generate	Generates a random seed.
init	Overloaded function.
seed	This static methods changes the seed of *egttools.Random*.

__init__(**args*, ***kwargs*)

__new__(***kwargs*)

static generate() → int
Generates a random seed.

The generated seed can be used to seed other pseudo-random generators, so that the initial state of the simulation can always be tracked and the simulation can be reproduced. This is very important both for debugging purposes as well as for scientific research. However, this approach should NOT be used in any cryptographic applications, it is NOT safe.

Returns A random seed which can be used to seed new random generators.

Return type int

static init(**args*, ***kwargs*)
Overloaded function.

1. init() -> egttools.numerical.Random

   This static method initializes the random seed generator from random_device and returns an instance of egttools.Random which is used to seed the random generators used across egttools.

   egttools.Random An instance of the random seed generator.

2. init(seed: int) -> egttools.numerical.Random

   This static method initializes the random seed generator from seed and returns an instance of *egttools.Random* which is used to seed the random generators used across *egttools*.

   seed [int] Integer value used to seed the random generator.

   egttools.Random An instance of the random seed generator.

static seed(*seed: int*) → None

This static methods changes the seed of `egttools.Random`.

Parameters **int** – The new seed for the `egttools.Random` module which is used to seed every other pseudo-random generation in the `egttools` package.

2.10 egttools.plotting

API reference documentation for the `plotting` submodule.

Functions

`draw_stationary_distribution`	Draws the markov chain for a given stationary distribution of monomorphic states.
`plot_gradient`	Creates a figure plotting the gradient of selection toghether with the saddle points, and the gradient arrows. :param x: vector containing the possible states in x axis. It must have the same length as gradient :param gradients: vector containing the gradient for each possible state :param saddle_points: vector containing all saddle points :param saddle_type: vector of booleans indicating whether or not the saddle point is stable :param gradient_direction: vector of points indicating the direction of the gradient between unstable and stable saddle points :param fig_title: a string containing the title of the figure :param xlabel: label for x axis :param figsize: a tuple indicating the size of the figure :param kwargs: you may pass an axis object :returns a figure object.

2.10.1 egttools.plotting.draw_stationary_distribution

draw_stationary_distribution(*strategies*, *drift*, *fixation_probabilities*, *stationary_distribution*, *max_displayed_label_letters=4*, *min_strategy_frequency=- 1*, *node_size=4000*, *font_size_node_labels=18*, *font_size_edge_labels=14*, *font_size_sd_labels=12*, *edge_width=2*, *figsize=(10, 10)*, *dpi=150*, *colors=None*, *ax=None*)

Draws the markov chain for a given stationary distribution of monomorphic states.

Parameters

- **strategies** (`List[str]`) – Strategies and array of string labels for each strategy present in the population.
- **drift** (`float`) – drift = 1/pop_size
- **fixation_probabilities** (`numpy.ndarray[float, 2]`) – A matrix specifying the fixation probabilities.
- **stationary_distribution** (`numpy.ndarray[float, 1]`) – An array containing the stationary distribution (probability of each state in the system).
- **max_displayed_label_letters** (`int`) – Maximum number of letters of the strategy labels contained in the `strategies` List to be displayed.

- **min_strategy_frequency** (*Optional[float]*) – Minimum frequency of a strategy (its probability given by the stationary distribution) to be shown in the Graph.
- **font_size_node_labels** (*Optional[int]*) – Font size of the labels displayed inside each node.
- **font_size_edge_labels** (*Optional[int]*) – Font size of the labels displayed in each edge (which contain the fixation probabilities).
- **font_size_sd_labels** (*Optional[int]*) – Font size of the labels displayed beside each node containing the value of the stationary distribution.
- **edge_width** (*Optional[int]*) – Width of the edge line.
- **figsize** (*Optional[Tuple[int, int]]*) – Size of the default figure (Only used if ax is not specified).
- **dpi** (*Optional[int]*) – Pixel density of the default plot
- **node_size** (*Optional[int]*) – Size of the nodes of the Graph to be plotted
- **colors** (*Optional[List[str]]*) – A list with the colors used to plot the nodes of the graph.
- **ax** (*Optional[plt.axis]*) – Axis on which to draw the graph.

Returns The graph depicting the Markov chain which represents the invasion dynamics.

Return type networkx.Graph

Notes

If there are too many strategies, this function may not only take a lot of time to generate the Graph, but it will also not be easy to visualize. Also, you should only use this function when ploting the invasion diagram assuming the small mutation limit of the replication dynamics (SML).

See also:

plot_gradient

Examples

```
>>> import matplotlib.pyplot as plt
>>> import egttools as egt
>>> strategies = [egt.behaviors.Cooperator(), egt.behaviors.Defector(), egt.
↪behaviors.TFT(),
...               egt.behaviors.Pavlov(), egt.behaviors.Random(), egt.behaviors.
↪GRIM()]
>>> strategy_labels = [strategy.type().replace("NFGStrategies::", '') for strategy␣
↪in strategies]
>>> T=4; R=2; P=1; S=0; Z= 100; beta=1
>>> A = np.array([
...     [P, T],
...     [S, R]
... ])
>>> game = egt.games.NormalFormGame(100, A, strategies)
>>> evolver = egt.analytical.StochDynamics(len(strategies), game.expected_payoffs(),
↪ Z)
```

(continues on next page)

(continued from previous page)

```
>>> sd = evolver.calculate_stationary_distribution(beta)
>>> transitions, fixation_probabilities = evolver.transition_and_fixation_
↪matrix(beta)
>>> fig, ax = plt.subplots(figsize=(5, 5), dpi=150)
>>> G = egt.plotting.draw_stationary_distribution(strategy_labels, 1/Z, fixation_
↪probabilities, sd,
...     node_size=2000, min_strategy_frequency=0.00001, ax=ax)
>>> plt.axis('off')
>>> plt.show() # display
```

2.10.2 egttools.plotting.plot_gradient

plot_gradient(*x*, *gradients*, *saddle_points*, *saddle_type*, *gradient_direction*, *fig_title=''*, *xlabel=''*, *figsize=(5, 4)*, ***kwargs*)

Creates a figure plotting the gradient of selection toghether with the saddle points, and the gradient arrows. :param x: vector containing the possible states in x axis. It must have the same length as gradient :param gradients: vector containing the gradient for each possible state :param saddle_points: vector containing all saddle points :param saddle_type: vector of booleans indicating whether or not the saddle point is stable :param gradient_direction: vector of points indicating the direction of the gradient

between unstable and stable saddle points

Parameters

- **fig_title** – a string containing the title of the figure
- **xlabel** – label for x axis
- **figsize** – a tuple indicating the size of the figure
- **kwargs** – you may pass an axis object

:returns a figure object

egttools.plotting.indicators	
egttools.plotting.simplex	

2.10.3 egttools.plotting.indicators

Functions

draw_stationary_distribution	Draws the markov chain for a given stationary distribution of monomorphic states.

continues on next page

Table 58 – continued from previous page

plot_gradient	Creates a figure plotting the gradient of selection toghether with the saddle points, and the gradient arrows. :param x: vector containing the possible states in x axis. It must have the same length as gradient :param gradients: vector containing the gradient for each possible state :param saddle_points: vector containing all saddle points :param saddle_type: vector of booleans indicating whether or not the saddle point is stable :param gradient_direction: vector of points indicating the direction of the gradient between unstable and stable saddle points :param fig_title: a string containing the title of the figure :param xlabel: label for x axis :param figsize: a tuple indicating the size of the figure :param kwargs: you may pass an axis object :returns a figure object.

egttools.plotting.indicators.draw_stationary_distribution

draw_stationary_distribution(*strategies*, *drift*, *fixation_probabilities*, *stationary_distribution*, *max_displayed_label_letters=4*, *min_strategy_frequency=- 1*, *node_size=4000*, *font_size_node_labels=18*, *font_size_edge_labels=14*, *font_size_sd_labels=12*, *edge_width=2*, *figsize=(10, 10)*, *dpi=150*, *colors=None*, *ax=None*)

Draws the markov chain for a given stationary distribution of monomorphic states.

Parameters

- **strategies** (*List[str]*) – Strategies and array of string labels for each strategy present in the population.
- **drift** (*float*) – drift = 1/pop_size
- **fixation_probabilities** (*numpy.ndarray[float, 2]*) – A matrix specifying the fixation probabilities.
- **stationary_distribution** (*numpy.ndarray[float, 1]*) – An array containing the stationary distribution (probability of each state in the system).
- **max_displayed_label_letters** (*int*) – Maximum number of letters of the strategy labels contained in the `strategies` List to be displayed.
- **min_strategy_frequency** (*Optional[float]*) – Minimum frequency of a strategy (its probability given by the stationary distribution) to be shown in the Graph.
- **font_size_node_labels** (*Optional[int]*) – Font size of the labels displayed inside each node.
- **font_size_edge_labels** (*Optional[int]*) – Font size of the labels displayed in each edge (which contain the fixation probabilities).
- **font_size_sd_labels** (*Optional[int]*) – Font size of the labels displayed beside each node containing the value of the stationary distribution.
- **edge_width** (*Optional[int]*) – Width of the edge line.
- **figsize** (*Optional[Tuple[int, int]]*) – Size of the default figure (Only used if ax is not specified).
- **dpi** (*Optional[int]*) – Pixel density of the default plot
- **node_size** (*Optional[int]*) – Size of the nodes of the Graph to be plotted

- **colors** (*Optional[List[str]]*) – A list with the colors used to plot the nodes of the graph.
- **ax** (*Optional[plt.axis]*) – Axis on which to draw the graph.

Returns The graph depicting the Markov chain which represents the invasion dynamics.

Return type networkx.Graph

Notes

If there are too many strategies, this function may not only take a lot of time to generate the Graph, but it will also not be easy to visualize. Also, you should only use this function when ploting the invasion diagram assuming the small mutation limit of the replication dynamics (SML).

See also:

plot_gradient

Examples

```
>>> import matplotlib.pyplot as plt
>>> import egttools as egt
>>> strategies = [egt.behaviors.Cooperator(), egt.behaviors.Defector(), egt.
↪behaviors.TFT(),
...               egt.behaviors.Pavlov(), egt.behaviors.Random(), egt.behaviors.
↪GRIM()]
>>> strategy_labels = [strategy.type().replace("NFGStrategies::", '') for strategy␣
↪in strategies]
>>> T=4; R=2; P=1; S=0; Z= 100; beta=1
>>> A = np.array([
...     [P, T],
...     [S, R]
... ])
>>> game = egt.games.NormalFormGame(100, A, strategies)
>>> evolver = egt.analytical.StochDynamics(len(strategies), game.expected_payoffs(),
↪ Z)
>>> sd = evolver.calculate_stationary_distribution(beta)
>>> transitions, fixation_probabilities = evolver.transition_and_fixation_
↪matrix(beta)
>>> fig, ax = plt.subplots(figsize=(5, 5), dpi=150)
>>> G = egt.plotting.draw_stationary_distribution(strategy_labels, 1/Z, fixation_
↪probabilities, sd,
...     node_size=2000, min_strategy_frequency=0.00001, ax=ax)
>>> plt.axis('off')
>>> plt.show() # display
```

egttools.plotting.indicators.plot_gradient

plot_gradient(*x*, *gradients*, *saddle_points*, *saddle_type*, *gradient_direction*, *fig_title=''*, *xlabel=''*, *figsize=(5, 4)*, ***kwargs*)

Creates a figure plotting the gradient of selection toghether with the saddle points, and the gradient arrows. :param x: vector containing the possible states in x axis. It must have the same length as gradient :param gradients: vector containing the gradient for each possible state :param saddle_points: vector containing all saddle points :param saddle_type: vector of booleans indicating whether or not the saddle point is stable :param gradient_direction: vector of points indicating the direction of the gradient

between unstable and stable saddle points

Parameters

- **fig_title** – a string containing the title of the figure
- **xlabel** – label for x axis
- **figsize** – a tuple indicating the size of the figure
- **kwargs** – you may pass an axis object

:returns a figure object

2.10.4 egttools.plotting.simplex

2.11 egttools.utils

This python module contains some utility functions to find saddle points and plot gradients in 2 player, 2 strategy games.

Functions

calculate_stationary_distribution	Calculates stationary distribution from a transition matrix of Markov chain.
combine	Outputs a generator that will generate an ordered list with the possible combinations of values with length.
find_saddle_type_and_gradient_direction	Finds whether a saddle point is stable or not.
get_payoff_function	Returns a function which gives the payoff of strategy i against strategy j.
transform_payoffs_to_pairwise	This function transform a payoff matrix in full format to a pairwise format.

2.11.1 egttools.utils.calculate_stationary_distribution

calculate_stationary_distribution(*transition_matrix*)

Calculates stationary distribution from a transition matrix of Markov chain.

The stationary distribution is the normalized eigenvector associated with the eigenvalue 1

Parameters **transition_matrix** (*numpy.ndarray*) – A 2 dimensional transition matrix

Returns A 1-dimensional vector containing the stationary distribution

Return type numpy.ndarray

2.11.2 egttools.utils.combine

combine(*values*, *length*)

Outputs a generator that will generate an ordered list with the possible combinations of values with length.

Each time the generator is called it will output a list of length :param length which contains a combinations of the elements in the list of values.

Parameters

- **values** (*List*) – elements to combine
- **length** (*int*) – size of the output

Returns A generator which outputs ordered combinations of value as a list of size length

Return type Generator

Examples

```
>>> for value in combine([1, 2], 2):
...     print(value)
[1, 1]
[2, 1]
[1, 2]
[2, 2]
```

2.11.3 egttools.utils.find_saddle_type_and_gradient_direction

find_saddle_type_and_gradient_direction(*gradient*, *saddle_points_idx*, *offset=0.01*)

Finds whether a saddle point is stable or not. And defines the direction of the gradient among stable and unstable points.

Parameters

- **gradient** (*{List[float], numpy.ndarray[float]}*) – array containing the gradient of selection for all states of the population
- **saddle_points_idx** (*{List[int], numpy.ndarray[int]}*) – array containing the saddle points indices
- **offset** (*float*) – offset for the gradient_directions, so that arrows don't overlap with point

Returns Tuple containing an array that indicates the type of saddle points and another array indicating the direction of the gradient between unstable and stable points

Return type Tuple[List[bool], List[float]]

2.11.4 egttools.utils.get_payoff_function

get_payoff_function(*strategy_i*, *strategy_j*, *nb_strategies*, *game*)

Returns a function which gives the payoff of strategy i against strategy j.

The returned function will return the payoff of strategy i given k individuals of strategy i and group_size - k j strategists.

Parameters

- **strategy_i** (*int*) – index of strategy i
- **strategy_j** (*int*) – index of strategy j
- **nb_strategies** (*int*) – Total number of strategies in the population.
- **game** (egttools.games.AbstractGame) – A game object which contains the method payoff which returns the payoff of a strategy given a group composition.

Returns A function which will return the payoff of strategy i given k individuals of strategy i and group_size - k j strategists.

Return type object

2.11.5 egttools.utils.transform_payoffs_to_pairwise

transform_payoffs_to_pairwise(*nb_strategies*, *game*)

This function transform a payoff matrix in full format to a pairwise format.

The transformation should only be done if it is possible to assume that there will always be at most 2 strategies in a group at a given time. An example of this would be when calculating the Small Mutation Limit (SML) of the Pairwise Moran Process. In this case, we do not need to know the payoffs for each strategy for any group composition (i.e., when there are more than 2 strategies in the group), but only for all possible combinations of each 2 strategies.

To be able to represent this in a nb_strategies x nb_strategies square matrix, we make each entry of the matrix a function, which will return the payoff of the strategy given k players adopting strategy i and N - k players adopting strategy j.

Parameters

- **nb_strategies** (*int*) – Number of strategies in the population
- **game** (egttools.games.AbstractGame) – A game object which contains the method payoff which returns the payoff of a strategy given a group composition.

Returns Returns the payoff matrix in shape nb_strategies x nb_strategies, and each entry of the payoff matrix is a function which will return the payoff of a strategy i against strategy j given a group composition with k members of strategy i and N - k members of strategy j.

Return type numpy.ndarray[numpy.float64[m,m]]

Classes

AbstractGame

2.11.6 egttools.utils.AbstractGame

class AbstractGame(*self:* egttools.numerical.games.AbstractGame) → None

Bases: pybind11_builtins.pybind11_object

Methods

calculate_fitness	Estimates the fitness for a player_type in the population with state :param strategies.
calculate_payoffs	Estimates the payoffs for each strategy and returns the values in a matrix.
nb_strategies	Number of different strategies playing the game.
payoff	Returns the payoff of a strategy given a group composition.
payoffs	returns the payoff matrix of the game.
play	Updates the vector of payoffs with the payoffs of each player after playing the game.
save_payoffs	Stores the payoff matrix in a txt file.
type	returns the type of game.

__init__(*self:* egttools.numerical.games.AbstractGame) → None

__new__(***kwargs*)

__str__(*self:* egttools.numerical.games.AbstractGame) → str

calculate_fitness(*self:* egttools.numerical.games.AbstractGame, *player_type:* *int*, *pop_size:* *int*, *strategies:* *numpy.ndarray[numpy.uint64[m, 1]]*) → float

Estimates the fitness for a player_type in the population with state :param strategies.

This function assumes that the player with strategy player_type is not included in the vector of strategy counts strategies.

Parameters

- **player_type** (*int*) – The index of the strategy used by the player.
- **pop_size** (*int*) – The size of the population.
- **strategies** (*numpy.ndarray[numpy.uint64[m, 1]]*) – A vector of counts of each strategy. The current state of the population.

Returns The fitness of the strategy in the population state given by strategies.

Return type float

calculate_payoffs(*self:* egttools.numerical.games.AbstractGame) → numpy.ndarray[numpy.float64[m, n]]

Estimates the payoffs for each strategy and returns the values in a matrix. Each row of the matrix represents a strategy and each column a game state. E.g., in case of a 2 player game, each entry a_ij gives the payoff

for strategy i against strategy j. In case of a group game, each entry a_ij gives the payoff of strategy i for game state j, which represents the group composition.

Returns A matrix with the expected payoffs for each strategy given each possible game state.

Return type numpy.ndarray[numpy.float64[m, n]]

nb_strategies(*self:* egttools.numerical.games.AbstractGame) → int
Number of different strategies playing the game.

payoff(*self:* egttools.numerical.games.AbstractGame, *strategy: int*, *group_composition: List[int]*) → float
Returns the payoff of a strategy given a group composition.

If the group composition does not include the strategy, the payoff should be zero.

Parameters

- **strategy** (*int*) – The index of the strategy used by the player.
- **group_composition** (*List[int]*) – List with the group composition. The structure of this list depends on the particular implementation of this abstract method.

Returns The payoff value.

Return type float

payoffs(*self:* egttools.numerical.games.AbstractGame) → numpy.ndarray[numpy.float64[m, n]]
returns the payoff matrix of the game.

play(*self:* egttools.numerical.games.AbstractGame, *group_composition: List[int]*, *game_payoffs: List[float]*) → None
Updates the vector of payoffs with the payoffs of each player after playing the game.

This method will run the game using the players and player types defined in :param group_composition, and will update the vector :param game_payoffs with the resulting payoff of each player.

Parameters

- **group_composition** (*List[int]*) – A list with counts of the number of players of each strategy in the group.
- **game_payoffs** (*List[float]*) – A list used as container for the payoffs of each player

save_payoffs(*self:* egttools.numerical.games.AbstractGame, *file_name: str*) → None
Stores the payoff matrix in a txt file.

Parameters **file_name** (*str*) – Name of the file in which the data will be stored.

type(*self:* egttools.numerical.games.AbstractGame) → str
returns the type of game.

CHAPTER

THREE

CITING EGTTOOLS

You may cite this repository in the following way:

```bibtex
@misc{Fernandez2020,
  author = {Fernández Domingos, Elias},
  title = {EGTTools: Toolbox for Evolutionary Game Theory},
  year = {2020},
  publisher = {GitHub},
  journal = {GitHub repository},
  howpublished = {\url{https://github.com/Socrats/EGTTools}},
  doi = {10.5281/zenodo.3687125}
}
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

INDEX

Symbols

A

C

D

E

F

G

I

L

M

N

P

R

S

T