Gambit Documentation

Release 16.6.0

The Gambit Project

Mar 25, 2026

CONTENTS

Gambit is a library of game theory software and tools for the construction and analysis of finite extensive and strategic games.

- **Users:** We recommend most newcomers install the PyGambit Python package and read the associated documentation, which includes tutorials and a complete API reference.
- **Contributors:** Please read the *code of conduct and contribution guidelines* before posting on GitHub.

Installing Gambit Quick installation with PyGambit: `pip install pygambit`

Install

PyGambit Explore tutorial notebooks and API reference docs.

PyGambit

Analysing games Compute equilibria and run econometric estimations.

Equilibrium computation

Catalog of games Browse a curated collection of game theory models.

Catalog of games

Graphical interface Interactively create, explore, and find equilibria of games.

GUI

Command-line interface Use Gambit's command-line tools for scripting.

CLI

Bugs and feature requests Report bugs and feature requests on GitHub.

GitHub issues

Developer docs Guides for developers & contributors to the package.

Developer docs

CHAPTER

ONE

INSTALL

Users installing Gambit have several options depending on their needs and their operating system. We recommended most new users install the PyGambit package and read the *PyGambit documentation*.

Developers & contributors should refer to the *developer docs* which contain guides for building Gambit from source and contributing to the project.

1.1 Installing PyGambit

PyGambit is available on PyPI. We recommend installing it into a Python virtual environment via *pip*:

```
pip install pygambit
```

Older releases can be installed by specifying the version number. Visit the Gambit releases page on GitHub for information on older versions.

1.2 Installing Gambit GUI & CLI tools

To install the Gambit *GUI* and *CLI tools*, visit the Gambit releases page on GitHub and download the appropriate installer or package for your operating system. Each release includes pre-built binaries for Windows, macOS, and Linux distributions, accessible under the "Assets" section of each release.

Install on macOS with disk image

1. **Download the .dmg installer:**

   Visit the Gambit releases page on GitHub and download the *.dmg* file for the version of Gambit you wish to install.

2. **Install the application:**

   Double click the *.dmg* file to mount it, then drag the Gambit application to your Applications folder.

> **Warning**
>
> You may need to adjust your macOS security settings to allow the installation of applications from unidentified developers.
>
> This can be done in `System Preferences > Security & Privacy` (see Apple's documentation for more details).
>
> If your administration privileges prevent this, try the Homebrew installation method below, or build from source as described in the *developer build instructions*.

Install on macOS via Homebrew

1. Check that you have Homebrew installed by running *brew –version*. If not, follow the instructions at https://brew.sh/.
2. Install Gambit using Homebrew:

```
brew install gambit
```

Warning

Homebrew installation has not been set up or tested by the Gambit development team.

Install on Linux or macOS from source tarball

1. **Download the source tarball:**

   Visit the Gambit releases page on GitHub and download the source tarball for the version of Gambit you wish to install.

2. **Extract the tarball:**

   Once downloaded, extract the tarball using the following command:

```
tar -xzf gambit-*.tar.gz
```

3. **Build and install Gambit:**

   Navigate to the extracted directory and run:

```
./configure
make
sudo make install
```

Note

Command-line options are available to modify the configuration process; do *./configure –help* for information. Of these, the option which may be most useful is to disable the build of the graphical interface.

By default Gambit will be installed in /usr/local. You can change this by replacing configure step with one of the form

```
`./configure --prefix=/your/path/here`
```

> **Warning**
>
> The graphical interface relies on external calls to other programs built in this process, especially for the computation of equilibria. It is strongly recommended that you install the Gambit executables to a directory in your path!

Install on Windows with installer

1. **Download the installer:**

   Visit the Gambit releases page on GitHub and download the *.msi*.

2. **Run the installer:**

   Double click the downloaded *.msi* file and follow the on-screen instructions to complete the installation.

CHAPTER

TWO

EQUILIBRIUM COMPUTATION

The table below summarizes the available PyGambit functions and corresponding Gambit CLI programs for algorithms for computing Nash equilibria.

Algorithm	Description	PyGambit function	CLI command
enumpure	Enumerate pure-strategy equilibria of a game	`pygambit.nash.enumpure_solve()`	*gambit-enumpure*
enummixed	Enumerate equilibria in a two-player game	`pygambit.nash.enummixed_solve()`	*gambit-enummixed*
enumpoly	Compute equilibria of a game using polynomial systems of equations	`pygambit.nash.enumpoly_solve()`	*gambit-enumpoly*
lp	Compute equilibria in a two-player constant-sum game via linear programming	`pygambit.nash.lp_solve()`	*gambit-lp*
lcp	Compute equilibria in a two-player game via linear complementarity	`pygambit.nash.lcp_solve()`	*gambit-lcp*
liap	Compute equilibria using function minimization	`pygambit.nash.liap_solve()`	*gambit-liap*
logit	Compute quantal response equilibria	`pygambit.nash.logit_solve()`	*gambit-logit*
simpdiv	Compute equilibria via simplicial subdivision	`pygambit.nash.simpdiv_solve()`	*gambit-simpdiv*
ipa	Compute equilibria using iterated polymatrix approximation	`pygambit.nash.ipa_solve()`	*gambit-ipa*
gnm	Compute equilibria using a global Newton method	`pygambit.nash.gnm_solve()`	*gambit-gnm*

2.1 enumpure

Searches for pure-strategy Nash or agent Nash equilibria.

2.2 enummixed

Computes Nash equilibria using extreme point enumeration.

In a two-player strategic game, the set of Nash equilibria can be expressed as the union of convex sets. This program generates all the extreme points of those convex sets. (Mangasarian [Man64]) This is a superset of the points generated by the path-following procedure of Lemke and Howson (see *lcp*). It was shown by Shapley [Sha74] that there are equilibria not accessible via the method in *lcp*, whereas the output of **enummixed** is guaranteed to return all the extreme points.

2.3 enumpoly

Computes Nash equilibria by solving systems of polynomial equations and inequalities.

This program searches for all Nash equilibria in a strategic game using a support enumeration approach. This approach computes all the supports which could, in principle, be the support of a Nash equilibrium. For each candidate support, it attempts to compute totally mixed equilibria on that support by successively subdividing the space of mixed strategy profiles or mixed behavior profiles (as appropriate). By using the fact that the equilibrium conditions imply a collection of equations and inequalities which can be expressed as multilinear polynomials, the subdivision constructed is such that each cell contains either no equilibria or exactly one equilibrium.

For strategic games, the program searches supports in the order proposed by Porter, Nudelman, and Shoham [PNS04]. For two-player games, this prioritises supports for which both players have the same number of strategies. For games with three or more players, this prioritises supports which have the fewest strategies in total. For many classes of games, this will tend to lower the average time until finding one equilibrium, as well as finding the second equilibrium (if one exists).

For extensive games, a support of actions equates to allowing positive probabilities over a subset of terminal nodes. The indifference conditions used are those for the sequence form defined on the projection of the game to that support of actions. A solution to these equations implies a probability distribution over terminal nodes. The algorithm then searches for a profile that is a Nash equilibrium that implements that probability distribution. If there exists at least one such profile, a sample one is returned. Note that for probability distributions which assign zero probability to some terminal nodes, it is generally the case that there are (infinitely) many such profiles. Subsequent analysis of unreached information sets can yield alternative profiles that specify different choices at unreached information sets while still satisfying the Nash equilibrium conditions.

2.4 lp

Computes a Nash equilibrium in a two-player game by solving a linear program. For extensive games, the program uses the sequence form formulation of Koller, Megiddo, and von Stengel [KolMegSte94].

While the set of equilibria in a two-player constant-sum strategic game is convex, this method will only identify one of the extreme points of that set.

2.5 lcp

Computes Nash equilibria of a two-player game by finding solutions to a linear complementarity problem. For extensive games, the program uses the sequence form representation of the extensive game, as defined by Koller, Megiddo, and von Stengel [KolMegSte94], and applies the algorithm developed by Lemke.

For strategic games, the program uses the method of Lemke and Howson [LemHow64]. In this case, the method will find all "accessible" equilibria, i.e., those that can be found as concatenations of Lemke-Howson paths that start at the artificial equilibrium. There exist strategic-form games for which some equilibria cannot be found by this method, i.e., some equilibria are inaccessible; see Shapley [Sha74].

In a two-player strategic game, the set of Nash equilibria can be expressed as the union of convex sets. This program will find extreme points of those convex sets. See *enummixed* for a method which is guaranteed to find all the extreme points for a strategic game.

2.6 liap

Computes approximate Nash equilibria using a function minimization approach.

This procedure searches for equilibria by generating random starting points and using conjugate gradient descent to minimize the Lyapunov function of the game. This is a nonnegative function which is zero exactly at strategy profiles

which are Nash equilibria.

Note that this procedure is not globally convergent. That is, it is not guaranteed to find all, or even any, Nash equilibria.

2.7 logit

Computes the principal branch of the (logit) quantal response correspondence.

The method is based on the procedure described in Turocy [Tur05] for strategic games and Turocy [Tur10] for extensive games. It uses standard path-following methods (as described in Allgower and Georg's "Numerical Continuation Methods") to adaptively trace the principal branch of the correspondence efficiently and securely.

The method used is a predictor-corrector method, which first generates a prediction using the differential equations describing the branch of the correspondence, followed by a corrector step which refines the prediction using Newton's method for finding a zero of a function. Two parameters control the operation of this tracing.

The algorithm accepts an initial step size for the predictor phase of the tracing. This step size is then dynamically adjusted based on the rate of convergence of Newton's method in the corrector step. If the convergence is fast, the step size is adjusted upward (accelerated); if it is slow, the step size is decreased (decelerated). The maximum acceleration (or deceleration) can be set as an argument. As described in Turocy [Tur05], this acceleration helps to efficiently trace the correspondence when it reaches its asymptotic phase for large values of the precision parameter lambda.

In extensive games, logit quantal response equilibria are not well-defined if an information set is not reached due to being the successor of chance moves with zero probability. In such games, the implementation treats the beliefs at such information sets as being uniform across all member nodes.

2.8 simpdiv

Computes approximations to Nash equilibria using a simplicial subdivision approach.

This program implements the algorithm of van der Laan, Talman, and van Der Heyden [VTH87]. The algorithm proceeds by constructing a triangulated grid over the space of mixed strategy profiles, and uses a path-following method to compute an approximate fixed point. This approximate fixed point can then be used as a starting point on a refinement of the grid. The program continues this process with finer and finer grids until locating a mixed strategy profile at which the maximum regret is small.

2.9 ipa

Computes Nash equilibria using an iterated polymatrix approximation approach developed by Govindan and Wilson [GovWil04]. This program is based on the Gametracer 0.2 implementation by Ben Blum and Christian Shelton.

The algorithm takes as a parameter a mixed strategy profile. This profile is interpreted as defining a ray in the space of games. The profile must have the property that, for each player, the most frequently played strategy must be unique.

2.10 gnm

Computes Nash equilibria using a global Newton method approach developed by Govindan and Wilson [GovWil03]. This program is based on the Gametracer 0.2 implementation by Ben Blum and Christian Shelton.

The algorithm takes as a parameter a mixed strategy profile. This profile is interpreted as defining a ray in the space of games. The profile must have the property that, for each player, the most frequently played strategy must be unique.

CHAPTER

THREE

PYGAMBIT

See installation instructions in the *Install* section.

For newcomers to Gambit, we recommend reading through the PyGambit tutorials, which demonstrate the API's key capabilities for analyzing and solving games. All of these tutorials assume a basic knowledge of programming in Python.

You can run the tutorials interactively as Jupyter notebooks, see local_tutorials.

3.1 New user tutorials

These tutorials assume no prior knowledge of Game Theory or the PyGambit API and provide detailed explanations of the concepts and code. They are numbered in the order they should be read.

3.1.1 1) Getting started with Gambit

In this tutorial, we'll demo the basic features of the Gambit library for game theory, using the `PyGambit` Python package.

This includes creating a `Game` object and using it to set up a strategic (normal) form game, the Prisoner's Dilemma, one of the most famous games in game theory.

We'll then use Gambit's built-in functions to analyze the game and find its Nash equilibria.

The Prisoner's Dilemma

The Prisoner's Dilemma is a classic example in game theory that illustrates why two rational individuals who cannot communicate might not cooperate, even if it appears that it is in their best interest to do so. After being caught by the police for committing a crime, the two prisoners are separately offered a deal:

- If both stay silent (cooperate), they get light sentences.
- If one defects (betrays the other) while the other stays silent, the defector goes free and the silent one gets a heavy sentence.
- If both defect, they both get moderate sentences.

Creating a strategic form game

Let's start by importing PyGambit and creating a game object. Since Prisoner's Dilemma is a strategic form game, it can be created in a tabular fashion with `Game.new_table`.

To do this, we need to know the number of players, which in Prisoner's Dilemma is 2, and the number of strategies for each player, which is in both cases is 2 (Cooperate and Defect). We'll define a list as long as the number of players, specifying the number of strategies for each player to pass into the `Game.new_table` function.

[1]:
```
import numpy as np

import pygambit as gbt
```

[2]:
```
n_strategies = [2, 2]
g = gbt.Game.new_table(n_strategies, title="Prisoner's Dilemma")
type(g)
```

[2]:
```
pygambit.gambit.Game
```

Now let's name the players and each of their possible strategies, in both cases "Cooperate" and "Defect".

Note: it's not necessary to specify labels for players and strategies when defining a game, however doing so makes the game easier to understand and work with.

[3]:
```
g.players[0].label = "Tom"
g.players[0].strategies[0].label = "Cooperate"
g.players[0].strategies[1].label = "Defect"

g.players[1].label = "Jerry"
g.players[1].strategies[0].label = "Cooperate"
g.players[1].strategies[1].label = "Defect"
```

Now let's assign payoffs for each of the game's possible outcomes, based on the standard payoffs for the Prisoner's Dilemma:

- Both players cooperate and receive the lightest sentence: `(-1, -1)`
- Tom cooperates, but Jerry defects (betrays Tom): `(0, -3)`
- Tom defects, Jerry cooperates: `(-3, 0)`
- Both defect: `(-2, -2)`

[4]:
```
# Both cooperate
g["Cooperate", "Cooperate"]["Tom"] = -1
g["Cooperate", "Cooperate"]["Jerry"] = -1

# Tom cooperates, Jerry defects
g["Cooperate", "Defect"]["Tom"] = -3
g["Cooperate", "Defect"]["Jerry"] = 0

# Tom defects, Jerry cooperates
g["Defect", "Cooperate"]["Tom"] = 0
g["Defect", "Cooperate"]["Jerry"] = -3

# Both defect
g["Defect", "Defect"]["Tom"] = -2
g["Defect", "Defect"]["Jerry"] = -2
```

[5]:
```
# View the payout matrix
g
```

[5]:
```
Game(title='Prisoner's Dilemma')
```

The payout matrix structure shows what in Game Theory is described as the "strategic form" (also "Normal-form") representation of a game.

The matrix presents the players' strategies and their expected payoff following their played strategies.

The strategic form assumes players choose their strategies simultaneously, and the outcome depends on the combination.

Creating games from arrays

The most direct way to create a strategic form game is via `Game.from_arrays()`.

This function takes one n-dimensional array per player, where n is the number of players in the game.

The arrays can be any object that can be indexed like an n-times-nested Python list; so, for example, numpy arrays can be used directly.

To create a two-player symmetric game, we can simply transpose the payoff matrix for the second player before passing to `Game.from_arrays()`.

[6]:
```
player1_payoffs = np.array([[-1, -3], [0, -2]])
player2_payoffs = np.transpose(player1_payoffs)

g1 = gbt.Game.from_arrays(
    player1_payoffs,
    player2_payoffs,
    title="Another Prisoner's Dilemma"
)

g1
```

[6]:
```
Game(title='Another Prisoner's Dilemma')
```

You can retrieve the players' payoff tables from a game object using the `Game.to_arrays()` method, which produces a list of numpy arrays representing the payoffs for each player.

The optional parameter `dtype` controls the data type of the payoffs in the generated arrays.

[7]:
```
tom_payoffs, jerry_payoffs = g.to_arrays(
    # dtype=float
)
print(tom_payoffs[0][0])
print(type(tom_payoffs[0][0]))
```

```
-1
<class 'pygambit.gambit.Rational'>
```

Computing the Nash equilibria in one line of code

We can use Gambit to compute the Nash equilibria for our Prisoner's Dilemma game in a single line of code. A Nash equilibrium describes a profile of strategies, one for each player, such that each player is maximizing their payoff given the strategies the other players are adopting.

For a two-player Normal-form game, let's use `enumpure_solve` to search for a pure-strategy Nash equilibria. The returned object will be a `NashComputationResult`.

[8]:
```
result = gbt.nash.enumpure_solve(g)
type(result)
```

[8]:
```
pygambit.nash.NashComputationResult
```

Let's inspect our result further to see how many equilibria were found.

[9]:
```
len(result.equilibria)
```

[9]:
```
1
```

For a given equilibria, we can then look at the "mixed strategy profile", which maps each strategy in a game to the corresponding probability with which that strategy is played.

[10]:
```
msp = result.equilibria[0]
msp
```

[10]: $[[0, 1], [0, 1]]$

[11]:
```
type(msp)
```

[11]:
```
pygambit.gambit.MixedStrategyProfileRational
```

The mixed strategy profile can show us the expected payoffs for each player when playing the strategies as specified by an equilibrium.

The profile `[[0,1],[0,1]]` indicates that both players' strategy is to play "Cooperate" with probability 0 and "Defect" with probability 1:

[12]:
```
for player in g.players:
    print(f"{player.label} plays the equilibrium strategy:")
    print(f"Probability of cooperating: {msp[player.label]['Cooperate']}")
    print(f"Probability of defecting: {msp[player.label]['Defect']}")
    print(f"Payoff: {msp.payoff(player.label)}")
    print()
```

```
Tom plays the equilibrium strategy:
Probability of cooperating: 0
Probability of defecting: 1
Payoff: -2

Jerry plays the equilibrium strategy:
Probability of cooperating: 0
Probability of defecting: 1
Payoff: -2
```

The equilibrium shows that both players are playing their dominant strategy, which is to defect. This is because defecting is the best response to the other player's strategy, regardless of what that strategy is.

Saving and reading strategic form games to and from file

You can use Gambit to save games to, and read from files. The specific format depends on whether the game is normal or extensive-form.

Here we'll save the Prisoner's Dilemma (Normal-form) to the `.nfg` format. Run these lines of code in new code cells if you're running the tutorial locally:

```
g.to_nfg("prisoners_dilemma.nfg")
```

You can easily restore the game object from file like so:

```
gbt.read_nfg("prisoners_dilemma.nfg")
```

3.1.2 2) Extensive-form games

In the first tutorial, we used Gambit to set up the Prisoner's Dilemma, an example of a normal (strategic) form game.

Gambit can also be used to set up extensive-form games; the game is represented as a tree, where each node represents a decision point for a player, and the branches represent the possible actions they can take.

Example: One-shot trust game with binary actions

Kreps (1990) introduced a game commonly referred to as the **trust game**. We will build a one-shot version of this game using Gambit's game transformation operations.

The game can be defined as follows:

- There are two players, a **Buyer** and a **Seller**.
- The Buyer moves first and has two actions, **Trust** or **Not trust**.
- If the Buyer chooses **Not trust**, then the game ends, and both players receive payoffs of `0`.
- If the Buyer chooses **Trust**, then the Seller has a choice with two actions, **Honor** or **Abuse**.
- If the Seller chooses **Honor**, both players receive payoffs of `1`;
- If the Seller chooses **Abuse**, the Buyer receives a payoff of `-1` and the Seller receives a payoff of `2`.

In addition to `pygambit`, this tutorial introduces the `draw_tree` package, which can be used to draw extensive form games in Python. If you're running this tutorial on your local machine, you'll need to install the requirements for draw_tree, which include LaTeX, in order to run the `draw_tree` cells. Another option for visualising extensive form games is to install the Gambit GUI and use it to load the EFG file generated at the end of this tutorial.

[1]:
```
from draw_tree import draw_tree

import pygambit as gbt
```

We create a game with an extensive representation using `Game.new_tree`:

[2]:
```
g = gbt.Game.new_tree(
    players=["Buyer", "Seller"],
    title="One-shot trust game, after Kreps (1990)"
)
```

The tree of the game contains just a root node, with no children:

[3]:
```
draw_tree(g)
```

[3]:

To extend a game from an existing terminal node, use `Game.append_move`. To begin with, the sole root node is the terminal node.

Here we extend the game from the root node by adding the first move for the "Buyer" player, creating two child nodes (one for each possible action).

[4]:
```
g.append_move(
    g.root,  # This is the node to append the move to
    player="Buyer",
```

(continues on next page)

(continued from previous page)

```
    actions=["Trust", "Not trust"]
)
```

[5]:
```
draw_tree(g)
```

[5]:

We can then also add the Seller's move in the situation after the Buyer chooses Trust:

[6]:
```
g.append_move(
    g.root.children["Trust"],
    player="Seller",
    actions=["Honor", "Abuse"]
)
```

[7]:
```
draw_tree(g)
```

[7]:

Now that we have the moves of the game defined, we add payoffs.

Payoffs are associated with an `Outcome`; each `Outcome` has a vector of payoffs, one for each player, and optionally an identifying text label.

First we add the outcome associated with the Seller proving themselves trustworthy:

[8]:
```
g.set_outcome(
    g.root.children["Trust"].children["Honor"],
    outcome=g.add_outcome(
        payoffs=[1, 1],
        label="Trustworthy"
    )
)
```

[9]:
```
draw_tree(g)
```

[9]:

Next, the outcome associated with the scenario where the Buyer trusts but the Seller does not return the trust:

[10]:
```
g.set_outcome(
    g.root.children["Trust"].children["Abuse"],
    outcome=g.add_outcome(
        payoffs=[-1, 2],
        label="Untrustworthy"
    )
)
```

[11]:
```
draw_tree(g)
```

[11]:

And, finally the outcome associated with the Buyer opting out of the interaction:

[12]:
```
g.set_outcome(
    g.root.children["Not trust"],
    g.add_outcome(
        payoffs=[0, 0],
```

(continues on next page)

(continued from previous page)

```
        label="Opt-out"
    )
)
```

[13]:
```
draw_tree(g)
```

[13]:

Nodes without an outcome attached are assumed to have payoffs of zero for all players.

Therefore, adding the outcome to this latter terminal node is not strictly necessary in Gambit, but it is useful to be explicit for readability.

Loading games from the catalog

Gambit includes a catalog of standard games that can be loaded directly by name. You can list all the available games like so:

[14]:
```
gbt.catalog.games()
```

[14]:

	Game	Title
0	bagwell1995	Bagwell (GEB 1995) commitment and (un)observab...
1	myerson1991/fig2_1	A simple Poker game
2	myerson1991/fig4_2	Myerson (1991) Figure 4.2
3	reiley2008/fig1	Stripped-down poker (Reiley et al 2008)
4	selten1975/fig1	Selten's horse (Selten IJGT 1975, Figure 1)
5	selten1975/fig2	Selten (IJGT 1975) Figure 2
6	selten1975/fig3	Selten (IJGT 1975) Figure 3
7	watson2013/exercise29_6	Princess Bride signaling game (from Watson)
8	watson2013/fig29_1	Job-market signaling game (version from Watson)

You can then load a specific game by its name. For example:

[15]:
```
g = gbt.catalog.load("selten1975/fig2")
draw_tree(g)
```

[15]:

Saving and reading extensive-form games to and from file

You can use Gambit to save games to, and read from files. The specific format depends on whether the game is normal or extensive-form.

Here we'll save the Trust game (extensive-form) to the `.efg` format. Run these lines of code in new code cells if you're running the tutorial locally:

```
g.to_efg("trust_game.efg")
```

You can easily restore the game object from file like so:

```
gbt.read_efg("trust_game.efg")
```

References

Kreps, D. (1990) "Corporate Culture and Economic Theory." In J. Alt and K. Shepsle, eds., *Perspectives on Positive Political Economy*, Cambridge University Press.

3.1.3 3) Stripped-down poker

In this tutorial, we'll create an extensive-form representation of a one-card poker game from *Reiley et al (2008)*, a classroom game under the name "stripped-down poker". This is perhaps the simplest interesting game with imperfect information.

We'll use "stripped-down poker" to demonstrate and explain the following with Gambit:

1. Setting up an extensive-form game with imperfect information using *information sets*
2. *Computing and interpreting Nash equilibria* and understanding mixed behaviour and mixed strategy profiles
3. *Acceptance criteria for Nash equilibria*

In our version of the game, there are two players, **Alice** and **Bob**, and a deck of cards, with equal numbers of **King** and **Queen** cards.

- The game begins with each player putting $1 in the pot.
  - A card is dealt at random to Alice.
  - Alice observes her card.
  - Bob does not observe the card.
- Alice then chooses either to **Bet** or to **Fold**.
  - If she chooses to Fold, Bob wins the pot and the game ends.
  - If she chooses to Bet, she adds another $1 to the pot.
- Bob then chooses either to **Call** or **Fold**.
  - If he chooses to Fold, Alice wins the pot and the game ends.
  - If he chooses to Call, he adds another $1 to the pot.
- There is then a showdown, in which Alice reveals her card.
  - If she has a King, then she wins the pot.
  - If she has a Queen, then Bob wins the pot.

In addition to `pygambit`, this tutorial uses the `draw_tree` package, which can be used to draw extensive form games in Python. If you're running this tutorial on your local machine, you'll need to install the requirements for draw_tree, which include LaTeX, in order to run the `draw_tree` cells. Another option for visualising extensive form games is to install the Gambit GUI and use it to load a saved EFG file.

[1]:
```
from draw_tree import draw_tree

import pygambit as gbt
```

Create the game with two players:

[2]:
```
g = gbt.Game.new_tree(
    players=["Alice", "Bob"],
    title="Stripped-Down Poker: a simple game of one-card poker from Reiley et al (2008).
↪"
)
```

In addition to the two named players, Gambit also instantiates a chance player.

[3]:
```
print(g.players["Alice"])
print(g.players["Bob"])
print(g.players.chance)
```

```
Player(game=Game(title='Stripped-Down Poker: a simple game of one-card poker from Reiley␣
↪et al (2008).'), label='Alice')
Player(game=Game(title='Stripped-Down Poker: a simple game of one-card poker from Reiley␣
↪et al (2008).'), label='Bob')
ChancePlayer(game=Game(title='Stripped-Down Poker: a simple game of one-card poker from␣
↪Reiley et al (2008).'))
```

Moves belonging to the chance player can be added in the same way as to other players.

At any new move created for the chance player, the action probabilities default to uniform randomization over the actions at the move.

The first step in this game is that Alice is dealt a card which could be a King or Queen, each with probability 1/2.

To simulate this in Gambit, we create a chance player move at the root node of the game:

[4]:
```
g.append_move(
    g.root,
    player=g.players.chance,
    actions=["King", "Queen"]  # By default, chance actions have equal probabilities
)
```

[5]:
```
draw_tree(g, color_scheme="gambit")
```

[5]:

Information sets

In this game, information structure is important. Alice knows her card, so the two nodes at which she has the move are part of different **information sets**.

We'll therefore need to append Alice's move separately for each of the root node's children, i.e. the scenarios where she has a King or a Queen. Let's now add both of these possible moves:

[6]:
```
for node in g.root.children:
    g.append_move(
        node,
        player="Alice",
        actions=["Bet", "Fold"]
    )
```

[7]:
```
draw_tree(g, color_scheme="gambit")
```

[7]:

The loop above causes each of the newly-appended moves to be in new information sets, reflecting the fact that Alice's decision depends on the knowledge of which card she holds.

In contrast, Bob does not know Alice's card, and therefore cannot distinguish between the two nodes at which he has to make his decision:

- Chance player chooses King, then Alice Bets: `g.root.children["King"].children["Bet"]`
- Chance player chooses Queen, then Alice Bets: `g.root.children["Queen"].children["Bet"]`

In other words, Bob's decision when Alice Bets with a Queen should be part of the same information set as Bob's decision when Alice Bets with a King.

To set this scenario up in Gambit, we'll need to add both possible moves as part of the same information set (represented in Gambit as an `Infoset`). This can be done by passing a list of nodes to the `append_move` method:

[8]:
```
g.append_move(
    [g.root.children["King"].children["Bet"], g.root.children["Queen"].children["Bet"]],
    player="Bob",
    actions=["Call", "Fold"]
)
```

[9]:
```
draw_tree(g, color_scheme="gambit")
```

[9]:

In game theory terms, this creates "imperfect information". Bob cannot distinguish between these two nodes in the game tree, so he must use the same same probabilities for Call vs. Fold in both situations.

This is crucial in games where players must make decisions without full knowledge of the state of the game.

Let's now set up the four possible payoff outcomes for the game. We'll label them according to player 1 (Alice):

[10]:
```
win_big = g.add_outcome([2, -2], label="Win Big")
win = g.add_outcome([1, -1], label="Win")
lose_big = g.add_outcome([-2, 2], label="Lose Big")
lose = g.add_outcome([-1, 1], label="Lose")
```

Finally, we should assign an outcome to each of the terminal nodes in the game tree:

[11]:
```
# Alice folds, Bob wins small
g.set_outcome(g.root.children["King"].children["Fold"], lose)
g.set_outcome(g.root.children["Queen"].children["Fold"], lose)

# Bob sees Alice Bet and calls, correctly believing she is bluffing, Bob wins big
g.set_outcome(g.root.children["Queen"].children["Bet"].children["Call"], lose_big)

# Bob sees Alice Bet and calls, incorrectly believing she is bluffing, Alice wins big
g.set_outcome(g.root.children["King"].children["Bet"].children["Call"], win_big)

# Bob does not call Alice's Bet, Alice wins small
g.set_outcome(g.root.children["King"].children["Bet"].children["Fold"], win)
g.set_outcome(g.root.children["Queen"].children["Bet"].children["Fold"], win)
```

[12]:
```
draw_tree(g, color_scheme="gambit")
```

[12]:

Computing and interpreting Nash equilibria

Since our one-card poker game has two players, we can use the `lcp_solve` algorithm in Gambit to compute a Nash equilibrium:

[13]:
```
result = gbt.nash.lcp_solve(g)
result
```

[13]:
```
NashComputationResult(method='lcp', rational=True, use_strategic=False,␣
↪equilibria=[[[[Rational(1, 1), Rational(0, 1)], [Rational(1, 3), Rational(2, 3)]],␣
↪[[Rational(2, 3), Rational(1, 3)]]]], parameters={'stop_after': None, 'max_depth':␣
↪None})
```

The result of the calculation is returned as a `NashComputationResult` object.

The set of equilibria found is reported in `NashComputationResult.equilibria`; in this case, this is a list of `MixedBehaviorProfile`'s.

For one-card poker, we expect to find a single equilibrium (one `MixedBehaviorProfile`):

[14]:
```
print("Number of equilibria found:", len(result.equilibria))
eqm = result.equilibria[0]
```

```
Number of equilibria found: 1
```

If we inspect the object type, we can see it's a `MixedBehaviorProfileRational` which is a subclass of `MixedBehaviorProfile` that uses rational numbers for probabilities:

[15]:
```
type(eqm)
```

[15]:
```
pygambit.gambit.MixedBehaviorProfileRational
```

A mixed behavior profile specifies, for each information set, the probability distribution over actions at that information set.

Indexing a mixed behaviour profile by a player gives a `MixedBehavior`, which specifies probability distributions at each of the player's information sets:

[16]:
```
type(eqm["Alice"])
```

[16]:
```
pygambit.gambit.MixedBehavior
```

[17]:
```
eqm["Alice"]
```

[17]: $\left[[1, 0], \left[\frac{1}{3}, \frac{2}{3}\right]\right]$

In this case, at Alice's first information set, the one at which she has the King, she always Bets.

At her second information set, where she has the Queen, she sometimes bluffs, raising with probability one-third.

The probability distribution at an information set is represented by a `MixedAction`.

`MixedBehavior.mixed_actions` iterates over these for the player:

[18]:
```
for infoset, mixed_action in eqm["Alice"].mixed_actions():
    print(
        f"At information set {infoset.number}, "
        f"Alice plays Bet with probability: {mixed_action['Bet']}"
        f" and Fold with probability: {mixed_action['Fold']}"
    )
```

```
At information set 0, Alice plays Bet with probability: 1 and Fold with probability: 0
At information set 1, Alice plays Bet with probability: 1/3 and Fold with probability: 2/
↪3
```

We can alternatively iterate through each of a player's actions like so:

[19]:
```
for action in g.players["Alice"].actions:
    print(
        f"At information set {action.infoset.number}, "
        f"Alice plays {action.label} with probability: {eqm[action]}"
    )
```

```
At information set 0, Alice plays Bet with probability: 1
At information set 0, Alice plays Fold with probability: 0
At information set 1, Alice plays Bet with probability: 1/3
At information set 1, Alice plays Fold with probability: 2/3
```

Now let's look at Bob's strategy:

[20]:
```
eqm["Bob"]
```

[20]: $[[\frac{2}{3}, \frac{1}{3}]]$

Bob Calls Alice's Bet two-thirds of the time. The label "Bet" is used in more than one information set for Alice, so in the above we had to specify information sets when indexing.

When there is no ambiguity, we can specify action labels directly. So for example, because Bob has only one action named "Call" in the game, we can extract the probability that Bob plays "Call" by:

[21]:
```
eqm["Bob"]["Call"]
```

[21]: $\frac{2}{3}$

Moreover, this is the only action with that label in the game, so we can index the profile directly using the action label without any ambiguity:

[22]:
```
eqm["Call"]
```

[22]: $\frac{2}{3}$

Because this is an equilibrium, Bob is indifferent between the two actions at his information set, meaning he has no reason to prefer one action over the other, in expectation, given Alice's expected strategy.

`MixedBehaviorProfile.action_value` returns the expected payoff of taking an action, conditional on reaching that action's information set:

[23]:
```
# Remember that Bob has a single information set
for action in g.players["Bob"].infosets[0].actions:
    print(
        f"When Bob plays {action.label} his expected payoff is {eqm.action_value(action)}
↪"
    )
```

```
When Bob plays Call his expected payoff is -1
When Bob plays Fold his expected payoff is -1
```

Bob's indifference between his actions arises because of his beliefs given Alice's strategy.

`MixedBehaviorProfile.belief` returns the probability of reaching a node, conditional on its information set being reached.

Recall that the two nodes in Bob's only information set are `g.root.children["King"].children["Bet"]` and `g.root.children["Queen"].children["Bet"]`):

[24]:
```
for node in g.players["Bob"].infosets[0].members:
    print(
        f"Bob's belief in reaching the {node.parent.prior_action.label} -> "
        f"{node.prior_action.label} node is: {eqm.belief(node)}"
    )
```

```
Bob's belief in reaching the King -> Bet node is: 3/4
Bob's belief in reaching the Queen -> Bet node is: 1/4
```

Bob believes that, conditional on Alice raising, there's a 3/4 chance that she has the King; therefore, the expected payoff to Calling is in fact -1 as computed.

`MixedBehaviorProfile.infoset_prob` returns the probability that an information set is reached:

[25]:
```
eqm.infoset_prob(g.players["Bob"].infosets[0])
```

[25]: $\frac{2}{3}$

The corresponding probability that a node is reached in the play of the game is given by `MixedBehaviorProfile.realiz_prob`, and the expected payoff to a player conditional on reaching a node is given by `MixedBehaviorProfile.node_value`:

[26]:
```
for node in g.players["Bob"].infosets[0].members:
    print(
        f"The probability that the node {node.parent.prior_action.label} -> "
        f"{node.prior_action.label} is reached is: {eqm.realiz_prob(node)}. ",
        f"Bob's expected payoff conditional on reaching this node is {eqm.node_value('Bob
↪', node)}"
    )
```

```
The probability that the node King -> Bet is reached is: 1/2.  Bob's expected payoff␣
↪conditional on reaching this node is -5/3
The probability that the node Queen -> Bet is reached is: 1/6.  Bob's expected payoff␣
↪conditional on reaching this node is 1
```

The overall expected payoff to a player given the behavior profile is returned by `MixedBehaviorProfile.payoff`:

[27]:
```
eqm.payoff("Alice")
```

[27]: $\frac{1}{3}$

[28]:
```
eqm.payoff("Bob")
```

[28]: $\frac{-1}{3}$

The equilibrium computed expresses probabilities in rational numbers.

Because the numerical data of games in Gambit *are represented exactly*, methods which are specialized to two-player games, `lp_solve`, `lcp_solve`, and `enummixed_solve`, can report exact probabilities for equilibrium strategy profiles.

This is enabled by default for these methods.

When a game has an extensive representation, equilibrium finding methods default to computing on that representation. It is also possible to compute using the strategic representation. `pygambit` transparently computes the reduced strategic form representation of an extensive game:

[29]:
```
[s.label for s in g.players["Alice"].strategies]
```

[29]:
```
['11', '12', '21', '22']
```

In the strategic form of this game, Alice has four strategies.

The generated strategy labels list the action numbers taken at each information set. For example, label '11' refers to the strategy gets dealt the King, then Bets.

We can therefore apply a method which operates on a strategic game to any game with an extensive representation.

[30]:
```
gnm_result = gbt.nash.gnm_solve(g)
gnm_result
```

[30]:
```
NashComputationResult(method='gnm', rational=False, use_strategic=True, equilibria=[[[0.
↪33333333333866677, 0.6666666666613335, 0.0, 0.0], [0.6666666666559997, 0.
↪33333333334440004]]], parameters={'perturbation': [[1.0, 0.0, 0.0, 0.0], [1.0, 0.0]],
↪'end_lambda': -10.0, 'steps': 100, 'local_newton_interval': 3, 'local_newton_maxits':␣
↪10})
```

`gnm_solve` can be applied to any game with any number of players, and uses a path-following process in floating-point arithmetic, so it returns profiles with probabilities expressed as floating-point numbers.

This method operates on the strategic representation of the game, so the returned results are of type `MixedStrategyProfile` (specifically `MixedStrategyProfileDouble`):

[31]:
```
gnm_eqm = gnm_result.equilibria[0]
type(gnm_eqm)
```

[31]:
```
pygambit.gambit.MixedStrategyProfileDouble
```

Indexing a `MixedStrategyProfile` by a player gives the probability distribution over that player's strategies only.

The expected payoff to a strategy is provided by `MixedStrategyProfile.strategy_value` and the overall expected payoff to a player is returned by `MixedStrategyProfile.payoff`:

[32]:
```
for player in g.players:
    print(
        f"{player.label}'s expected payoffs playing:"
    )
    for strategy in player.strategies:
        print(
            f"Strategy {strategy.label}: {gnm_eqm.strategy_value(strategy):.4f}"
        )
    print(
        f"{player.label}'s overall expected payoff: {gnm_eqm.payoff(player):.4f}"
    )
    print()
```

```
Alice's expected payoffs playing:
Strategy 11: 0.3333
Strategy 12: 0.3333
Strategy 21: -1.0000
Strategy 22: -1.0000
Alice's overall expected payoff: 0.3333

Bob's expected payoffs playing:
Strategy 1: -0.3333
Strategy 2: -0.3333
Bob's overall expected payoff: -0.3333
```

When a game has an extensive representation, we can convert freely between a mixed strategy profile and the corresponding mixed behaviour profile representation of the same strategies using `MixedStrategyProfile.as_behavior` and `MixedBehaviorProfile.as_strategy`.

- A mixed **strategy** profile maps each strategy in a game to the corresponding probability with which that strategy is played.

- A mixed **behaviour** profile maps each action at each information set in a game to the corresponding probability with which the action is played, conditional on that information set being reached.

Let's convert the equilibrium we found using `gnm_solve` to a mixed behaviour profile and iterate through the players actions to show their expected payoffs, comparing as we go with the payoffs found by `lcp_solve`:

[33]:
```
for player in g.players:
    print(
        f"{player.label}'s expected payoffs:"
    )
    for action in player.actions:
        print(
            f"At information set {action.infoset.number}, "
            f"when playing {action.label} - "
            f"gnm: {gnm_eqm.as_behavior().action_value(action):.4f}"
            f", lcp: {str(eqm.action_value(action))}"
        )
    print()
```

```
Alice's expected payoffs:
At information set 0, when playing Bet - gnm: 1.6667, lcp: 5/3
At information set 0, when playing Fold - gnm: -1.0000, lcp: -1
At information set 1, when playing Bet - gnm: -1.0000, lcp: -1
At information set 1, when playing Fold - gnm: -1.0000, lcp: -1

Bob's expected payoffs:
At information set 0, when playing Call - gnm: -1.0000, lcp: -1
At information set 0, when playing Fold - gnm: -1.0000, lcp: -1
```

Acceptance criteria for Nash equilibria

Some methods for computing Nash equilibria operate using floating-point arithmetic and/or generate candidate equilibrium profiles using methods which involve some form of successive approximations. The outputs of these methods therefore are in general ε-equilibria, for some positive ε.

ε-equilibria (from Wikipedia):

> In game theory, an epsilon-equilibrium, or near-Nash equilibrium, is a strategy profile that approximately satisfies the condition of Nash equilibrium. In a Nash equilibrium, no player has an incentive to change his behavior. In an approximate Nash equilibrium, this requirement is weakened to allow the possibility that a player may have a small incentive to do something different.
>
> Given a game and a real non-negative parameter ε, a strategy profile is said to be an ε-equilibrium if it is not possible for any player to gain more than ε in expected payoff by unilaterally deviating from his strategy. Every Nash Equilibrium is an ε-equilibrium where $\varepsilon = 0$.

To provide a uniform interface across methods, where relevant Gambit provides a parameter `maxregret`, which specifies the acceptance criterion for labeling the output of the algorithm as an equilibrium. This parameter is interpreted *proportionally* to the range of payoffs in the game. Any profile returned as an equilibrium is guaranteed to be an ε-equilibrium, for ε no more than `maxregret` times the difference of the game's maximum and minimum payoffs.

As an example, consider solving our one-card poker game using `logit_solve`. The range of the payoffs in this game is 4 (from +2 to -2):

[34]:
```
g.max_payoff, g.min_payoff
```

[34]:
```
(Rational(2, 1), Rational(-2, 1))
```

`logit_solve` is a globally-convergent method, in that it computes a sequence of profiles which is guaranteed to have a subsequence that converges to a Nash equilibrium.

The default value of `maxregret` for this method is set at `1e-8`:

[35]:
```
logit_solve_result = gbt.nash.logit_solve(g, maxregret=1e-8)
len(logit_solve_result.equilibria)
```

[35]:
```
1
```

[36]:
```
ls_eqm = logit_solve_result.equilibria[0]
ls_eqm.max_regret()
```

[36]:
```
2.6582744783176793e-08
```

The value of `MixedBehaviorProfile.max_regret` of the computed profile exceeds `1e-8` measured in terms of payoffs of the game. However, when considered relative to the scale of the game's payoffs, we see it is less than `1e-8` of the payoff range, as requested:

[37]:
```
ls_eqm.max_regret() / (g.max_payoff - g.min_payoff)
```

[37]:
```
6.645686195794198e-09
```

In general, for globally-convergent methods especially, there is a tradeoff between precision and running time.

We could instead ask only for an ε-equilibrium with a (scaled) ε of no more than `1e-4`:

[38]:
```
(
    gbt.nash.logit_solve(g, maxregret=1e-4).equilibria[0]
    .max_regret() / (g.max_payoff - g.min_payoff)
)
```

[38]:
```
6.265863445534259e-05
```

The tradeoff comes from some methods being slow to converge on some games, making it useful instead to get a more coarse approximation to an equilibrium (higher `maxregret` value) which is faster to calculate:

[39]:
```
%%time
gbt.nash.logit_solve(g, maxregret=1e-4)
```

```
CPU times: user 8.14 ms, sys: 22 s, total: 8.16 ms
Wall time: 8.14 ms
```

[39]:
```
NashComputationResult(method='logit', rational=False, use_strategic=False,␣
↪equilibria=[[[[1.0, 0.0], [0.3338351656285656, 0.666164834417892]], [[0.
↪6670407651644306, 0.3329592348608147]]]], parameters={'first_step': 0.03, 'max_accel':␣
↪1.1})
```

[40]:
```
%%time
gbt.nash.logit_solve(g, maxregret=1e-8)
```

```
CPU times: user 14.6 ms, sys: 945 s, total: 15.6 ms
Wall time: 15.5 ms
```

[40]:
```
NashComputationResult(method='logit', rational=False, use_strategic=False,␣
↪equilibria=[[[[1.0, 0.0], [0.33333338649882943, 0.6666666135011707]], [[0.
```

(continues on next page)

(continued from previous page)

```
↪6666667065407631, 0.3333332934592369]]]], parameters={'first_step': 0.03, 'max_accel':␣
↪1.1})
```

The convention of expressing `maxregret` scaled by the game's payoffs standardises the behavior of methods across games.

For example, consider solving the poker game instead using `liap_solve()`.

[41]:
```
(
    gbt.nash.liap_solve(g.mixed_strategy_profile(), maxregret=1e-1)
    .equilibria[0].max_regret() / (g.max_payoff - g.min_payoff)
)
```

[41]:
```
0.03657804319443689
```

If, instead, we double all payoffs, the output of the method is unchanged:

[42]:
```
for outcome in g.outcomes:
    outcome["Alice"] = outcome["Alice"] * 2
    outcome["Bob"] = outcome["Bob"] * 2

(
    gbt.nash.liap_solve(g.mixed_strategy_profile(), maxregret=1e-1)
    .equilibria[0].max_regret() / (g.max_payoff - g.min_payoff)
)
```

[42]:
```
0.03657804319443689
```

Representation of numerical data of a game

Payoffs to players and probabilities of actions at chance information sets are specified as numbers. Gambit represents the numerical values in a game in exact precision, using either decimal or rational representations.

To illustrate, consider a trivial game which just has one move for the chance player:

[43]:
```
small_game = gbt.Game.new_tree()
small_game.append_move(small_game.root, small_game.players.chance, ["a", "b", "c"])
[act.prob for act in small_game.root.infoset.actions]
```

[43]:
```
[Rational(1, 3), Rational(1, 3), Rational(1, 3)]
```

The default when creating a new move for chance is that all actions are chosen with equal probability. These probabilities are represented as rational numbers, using `pygambit`'s `Rational` class, which is derived from Python's `fractions.Fraction`.

Numerical data can be set as rational numbers. Here we update the chance action probabilities with `Rational` numbers:

[44]:
```
small_game.set_chance_probs(
    small_game.root.infoset,
    [gbt.Rational(1, 4), gbt.Rational(1, 2), gbt.Rational(1, 4)]
)
[act.prob for act in small_game.root.infoset.actions]
```

[44]:
```
[Rational(1, 4), Rational(1, 2), Rational(1, 4)]
```

Numerical data can also be explicitly specified as decimal numbers:

[45]:
```
small_game.set_chance_probs(
    small_game.root.infoset,
    [gbt.Decimal(".25"), gbt.Decimal(".50"), gbt.Decimal(".25")]
)
[act.prob for act in small_game.root.infoset.actions]
```

[45]:
```
[Decimal('0.25'), Decimal('0.50'), Decimal('0.25')]
```

Although the two representations above are mathematically equivalent, `pygambit` remembers the format in which the values were specified.

Expressing rational or decimal numbers as above is verbose and tedious. `pygambit` offers a more concise way to express numerical data in games: when setting numerical game data, `pygambit` will attempt to convert text strings to their rational or decimal representation. The above can therefore be written more compactly using string representations:

[46]:
```
small_game.set_chance_probs(small_game.root.infoset, ["1/4", "1/2", "1/4"])
[act.prob for act in small_game.root.infoset.actions]
```

[46]:
```
[Rational(1, 4), Rational(1, 2), Rational(1, 4)]
```

[47]:
```
small_game.set_chance_probs(small_game.root.infoset, [".25", ".50", ".25"])
[act.prob for act in small_game.root.infoset.actions]
```

[47]:
```
[Decimal('0.25'), Decimal('0.50'), Decimal('0.25')]
```

As a further convenience, `pygambit` will accept Python `int` and `float` values. `int` values are always interpreted as `Rational` values.

`pygambit` attempts to render `float` values in an appropriate `Decimal` equivalent. In the majority of cases, this creates no problems. For example:

[48]:
```
small_game.set_chance_probs(small_game.root.infoset, [.25, .50, .25])
[act.prob for act in small_game.root.infoset.actions]
```

[48]:
```
[Decimal('0.25'), Decimal('0.5'), Decimal('0.25')]
```

However, rounding can cause difficulties when attempting to use `float` values to represent values which do not have an exact decimal representation:

[49]:
```
try:
    small_game.set_chance_probs(small_game.root.infoset, [1/3, 1/3, 1/3])
except ValueError as e:
    print("ValueError:", e)
```

```
ValueError: set_chance_probs(): must specify non-negative probabilities that sum to one
```

This behavior can be slightly surprising, especially in light of the fact that in Python:

[50]:
```
1/3 + 1/3 + 1/3
```

[50]:
```
1.0
```

In checking whether these probabilities sum to one, `pygambit` first converts each of the probabilities to a `Decimal` representation, via the following method:

[51]:
```
gbt.Decimal(str(1/3))
```

[51]:
```
Decimal('0.3333333333333333')
```

...and the sum-to-one check then fails because:

[52]:
```
gbt.Decimal(str(1/3)) + gbt.Decimal(str(1/3)) + gbt.Decimal(str(1/3))
```

[52]:
```
Decimal('0.9999999999999999')
```

Setting payoffs for players also follows the same rules. Representing probabilities and payoffs exactly is essential, because `pygambit` offers (in particular for two-player games) the possibility of computation of equilibria exactly, because the Nash equilibria of any two-player game with rational payoffs and chance probabilities can be expressed exactly in terms of rational numbers.

It is therefore advisable always to specify the numerical data of games either in terms of `Decimal` or `Rational` values, or their string equivalents. It is safe to use `int` values, but `float` values should be used with some care to ensure the values are recorded as intended.

References

Reiley, David H., Michael B. Urbancic and Mark Walker. (2008) "Stripped-down poker: A classroom game with signaling and bluffing." *The Journal of Economic Education* 39(4): 323-341.

3.1.4 4) Creating publication-ready game images

[1]:
```
from draw_tree import draw_tree

import pygambit as gbt
```

Using a combination of `pygambit` and the Gambit project's LaTeX graphics package `draw_tree`, we can generate publication quality images for games with just a few lines of code.

This tutorial will demonstrate the key functionality of `draw_tree` when used for games built with `pygambit`, using an example game derived from the Gambit catalog. First, let's load the game:

[2]:
```
g = gbt.read_efg("../../contrib/games/2smp.efg")
```

Now let's see how `draw_tree` renders it with default settings:

[3]:
```
draw_tree(g)
```

[3]:

Already this looks good, but perhaps it would look neater if the terminal nodes were all extended to the bottom of the image for consistency. To achieve this, set `shared_terminal_depth=True`:

[4]:
```
draw_tree(
    g,
    shared_terminal_depth=True
)
```

[4]:

This image is quite large, but the game isn't overly complex, so we can reduce the size with the `scale_factor`:

[5]:
```
draw_tree(
    g,
    shared_terminal_depth=True,
    scale_factor=0.5
)
```

[5]:

Perhaps the size of the image is roughly correct, but we want to reduce the distance between node levels; we can scale the level spacing with `level_scaling`:

[6]:
```
draw_tree(
    g,
    shared_terminal_depth=True,
    scale_factor=0.5,
    level_scaling=0.75
)
```

[6]:

By default, Gambit's layout algorithm has split nodes at the same level (depth) of the game across sublevels. For this game, it will probably look better to `sublevel_scaling` to zero:

[7]:
```
draw_tree(
    g,
    shared_terminal_depth=True,
    scale_factor=0.5,
    level_scaling=0.75,
    sublevel_scaling=0
)
```

[7]:

The player labels are looking a little cramped. Let's adjust the width by increasing the `width_scaling`:

[8]:
```
draw_tree(
    g,
    shared_terminal_depth=True,
    scale_factor=0.5,
    level_scaling=0.75,
    sublevel_scaling=0,
    width_scaling=1.25
)
```

[8]:

If we want to use color in our image, we can set the `color_scheme`. Here let's use the "gambit" color scheme:

[9]:
```
draw_tree(
    g,
    shared_terminal_depth=True,
    scale_factor=0.5,
    level_scaling=0.75,
    sublevel_scaling=0,
    width_scaling=1.25,
    color_scheme="gambit"
)
```

[9]:

An advantage to using a color scheme is that nodes no longer require player labels, which are handled by the legend, de-cluttering the image further.

Let's make our image more striking by increasing the `edge_thickness`:

[10]:
```
draw_tree(
    g,
```

(continues on next page)

(continued from previous page)

```
    shared_terminal_depth=True,
    scale_factor=0.5,
    level_scaling=0.75,
    sublevel_scaling=0,
    width_scaling=1.25,
    color_scheme="gambit",
    edge_thickness=1.5
)
```

[10]:

One final adjustment we could make would be to adjust the positioning of action labels on the image. This can be helpful in cases where the action labels overlap visually with information sets boundaries or other features. The default value of 0.5 places the labels halfway along the edges, which for this game looks about right, but we can change this by setting `action_label_position`:

[11]:
```
draw_tree(
    g,
    shared_terminal_depth=True,
    scale_factor=0.5,
    level_scaling=0.75,
    sublevel_scaling=0,
    width_scaling=1.25,
    color_scheme="gambit",
    edge_thickness=1.5,
    action_label_position=0.6
)
```

[11]:

Saving images

Once we are happy with our image, we can save it as a Tex file, or generate a PNG or PDF with the rendered image. Each of the following functions takes the same arguments as the `draw_tree` examples above. Setting the `save_to` argument will determine where the `.ef` file used by `draw_tree` to preserve layout information is saved, as well as the output image or Tex file:

[12]:
```
draw_tree(
    g,
    shared_terminal_depth=True,
    scale_factor=0.5,
    level_scaling=0.75,
    sublevel_scaling=0,
    width_scaling=1.25,
    color_scheme="gambit",
    edge_thickness=1.5,
    action_label_position=0.6,
    save_to="2smp"  # Creates 2smp.ef
)
```

[12]:

Save to TeX

```
from draw_tree generate_tex
generate_tex(
    g,
    shared_terminal_depth=True,
    scale_factor=0.5,
    level_scaling=0.75,
    sublevel_scaling=0,
    width_scaling=1.25,
    color_scheme="gambit",
    edge_thickness=1.5,
    action_label_position=0.6,
    save_to="2smp"  # Creates 2smp.ef and 2smp.tex
)
```

Save to PDF

```
from draw_tree import generate_pdf
generate_pdf(
    g,
    shared_terminal_depth=True,
    scale_factor=0.5,
    level_scaling=0.75,
    sublevel_scaling=0,
    width_scaling=1.25,
    color_scheme="gambit",
    edge_thickness=1.5,
    action_label_position=0.6,
    save_to="2smp"  # Creates 2smp.ef and 2smp.pdf
)
```

Save to PNG

```
from draw_tree import generate_png
generate_png(
    g,
    shared_terminal_depth=True,
    scale_factor=0.5,
    level_scaling=0.75,
    sublevel_scaling=0,
    width_scaling=1.25,
    color_scheme="gambit",
    edge_thickness=1.5,
    action_label_position=0.6,
    save_to="2smp"  # Creates 2smp.ef and 2smp.png
)
```

Further adjustments to game images

If your image requires further adjustments, you can manually edit your game's `.ef` file. You can find information on EF format specs in the draw_tree docs.

EF files that are manually created or (exported from Game Theory Explorer can be drawn by `draw_tree` with the same functions explored in this tutorial, but you should drop the formatting parameters:

```
draw_tree('path/to/game.ef')
generate_tex('path/to/game.ef')
generate_pdf('path/to/game.ef')
generate_png('path/to/game.ef')
```

3.2 Advanced user tutorials

These tutorials assume some familiarity with the PyGambit API and Game Theory terminology and concepts including:

- Nash equilibria
- Pure and mixed strategies
- Simplex representations of available strategies
- Logit quantal response equilibrium (LQRE) correspondence

Advanced tutorials:

3.2.1 Generating starting points for algorithms

In the previous tutorial, we demonstrated how to calculate the Nash equilibria of a game set up using Gambit and interpret the `MixedStrategyProfile` or `MixedBehaviorProfile` objects returned by the solver. In this tutorial, we will demonstrate how to use a `MixedStrategyProfile` or `MixedBehaviorProfile` as an initial condition, a starting point, for some methods of computing Nash equilibria. The equilibria found will depend on which starting point is selected.

To facilitate generating starting points, Gambit's `Game` class provides the methods `random_strategy_profile` and `random_behavior_profile`, to generate profiles which are drawn from the uniform distribution on the product of simplices. In other words, the profiles are sampled from a uniform distribution so that each possible mixed strategy profile (or mixed behaviour profile) is equally likely to be selected.

As an example, we consider a three-player game from McKelvey and McLennan (1997), in which each player has two strategies. This game has nine equilibria in total, and in particular has two totally mixed Nash equilibria, which is the maximum possible number of regular totally mixed equilbria in games of this size.

Pure and mixed strategies:

- **Pure strategy**: A player chooses the action with probability 1 (always picks the same move)
- **Mixed strategy**: A player assigns probabilities to their available actions (some actions may have probability 0)
- **Totally mixed strategy**: Mixed strategy where every available action is chosen with strictly positive probability (no action has probability 0)

[1]:
```
import numpy as np

import pygambit as gbt

g = gbt.read_nfg("../../2x2x2.nfg")
g
```

We first consider finding Nash equilibria in this game using `liap_solve`. If we run this method starting from the centroid (uniform randomization across all strategies for each player), `liap_solve` finds one of the totally-mixed equilibria. Without providing a list to `Game.mixed_strategy_profile`, the method will return the centroid mixed strategy profile.

[ ]:
```
centroid_start = g.mixed_strategy_profile()
centroid_start
```

[ ]:
```
gbt.nash.liap_solve(centroid_start).equilibria[0]
```

As you can see, in this totally mixed strategy equilibrium, no action has probability 0.

Which equilibrium is found depends on the starting point. With a different starting point, we can find, for example, one of the pure-strategy equilibria.

[ ]:
```
new_start = g.mixed_strategy_profile([[.9, .1], [.9, .1], [.9, .1]])
new_start
```

[ ]:
```
gbt.nash.liap_solve(new_start).equilibria[0]
```

To search for more equilibria, we can instead generate strategy profiles at random.

[ ]:
```
random_start = g.random_strategy_profile()
random_start
```

[ ]:
```
gbt.nash.liap_solve(random_start).equilibria[0]
```

Note that methods which take starting points do record the starting points used in the result object returned. However, the random profiles which are generated will differ in different runs of a program.

To support making the generation of random strategy profiles reproducible, and for finer-grained control of the generation of these profiles if desired, `Game.random_strategy_profile` and `Game.random_behavior_profile` optionally take a `numpy.random.Generator` object, which is used as the source of randomness for creating the profile.

[ ]:
```
gen = np.random.default_rng(seed=1234567890)
p1 = g.random_strategy_profile(gen=gen)
gen = np.random.default_rng(seed=1234567890)
p2 = g.random_strategy_profile(gen=gen)
p1 == p2
```

When creating profiles in which probabilities are represented as floating-point numbers, `Game.random_strategy_profile` and `Game.random_behavior_profile` internally use the Dirichlet distribution for each simplex to generate correctly uniform sampling over probabilities. However, in some applications generation of random profiles with probabilities as rational numbers is desired.

For example, `simpdiv_solve` takes such a starting point, because it operates by successively refining a triangulation over the space of mixed strategy profiles. `Game.random_strategy_profile` and `Game.random_behavior_profile` both take an optional parameter `denom` which, if specified, generates a profile in which probabilities are generated uniformly from the grid in each simplex in which all probabilities have denominator `denom`.

These can then be used in conjunction with `simpdiv_solve` to search for equilibria from different starting points.

[ ]:
```
gen = np.random.default_rng(seed=1234567890)
rsp = g.random_strategy_profile(denom=10, gen=gen)
rsp
```

[ ]:
```
gbt.nash.simpdiv_solve(rsp).equilibria[0]
```

[ ]:
```
rsp1 = g.random_strategy_profile(denom=10, gen=gen)
rsp1
```

[ ]:
```
gbt.nash.simpdiv_solve(rsp1).equilibria[0]
```

[ ]:
```
rsp2 = g.random_strategy_profile(denom=10, gen=gen)
rsp2
```

[ ]:
```
gbt.nash.simpdiv_solve(rsp2).equilibria[0]
```

3.2.2 Quantal response equilibria

Gambit implements the idea of McKPal95 and McKPal98 to compute Nash equilibria via path-following a branch of the logit quantal response equilibrium (LQRE) correspondence using the function `logit_solve`. As an example, we will consider an asymmetric matching pennies game from Och95 as analyzed in McKPal95.

[1]:
```
import pygambit as gbt
```

[2]:
```
g = gbt.Game.from_arrays(
        [[1.1141, 0], [0, 0.2785]],
        [[0, 1.1141], [1.1141, 0]],
        title="Ochs (1995) asymmetric matching pennies as transformed in McKelvey-
↪Palfrey (1995)"
)
gbt.nash.logit_solve(g).equilibria[0]
```

[2]:
```
[[0.5000000234106035, 0.49999997658939654], [0.19998563837426647, 0.8000143616257336]]
```

`logit_solve` returns only the limiting (approximate) Nash equilibrium found. Profiles along the QRE correspondence are frequently of interest in their own right. Gambit offers several functions for more detailed examination of branches of the QRE correspondence.

The function `logit_solve_branch` uses the same procedure as `logit_solve`, but returns a list of LQRE profiles computed along the branch instead of just the limiting approximate Nash equilibrium.

[3]:
```
qres = gbt.qre.logit_solve_branch(g)
len(qres)
```

[3]:
```
193
```

[4]:
```
qres[0].profile
```

[4]:
```
[[0.5, 0.5], [0.5, 0.5]]
```

[5]:
```
qres[5].profile
```

[5]:
```
[[0.5182276540742868, 0.4817723459257562], [0.49821668880066783, 0.5017833111993909]]
```

`logit_solve_branch` uses an adaptive step size heuristic to find points on the branch. The parameters `first_step` and `max_accel` are used to adjust the initial step size and the maximum rate at which the step size changes adaptively. The step size used is computed as the distance traveled along the path, and, importantly, not the distance as measured by changes in the precision parameter lambda. As a result the lambda values for which profiles are computed cannot be controlled in advance.

In some situations, the LQRE profiles at specified values of lambda are of interest. For this, Gambit provides `logit_solve_lambda`. This function provides accurate values of strategy profiles at one or more specified values of lambda.

[6]:
```
qres = gbt.qre.logit_solve_lambda(g, lam=[1, 2, 3])
qres[0].profile
```

[6]: [[0.5867840364385154, 0.4132159635614846], [0.45180703169971026, 0.5481929683002897]]

[7]:
```
qres[1].profile
```

[7]: [[0.6175219458400859, 0.3824780541599141], [0.3719816648492249, 0.6280183351507752]]

[8]:
```
qres[2].profile
```

[8]: [[0.6168968501329284, 0.3831031498670716], [0.31401636202001226, 0.6859836379799878]]

LQRE are frequently taken to data by using maximum likelihood estimation to find the LQRE profile that best fits an observed profile of play. This is provided by the function `logit_estimate`. We replicate the analysis of a block of the data from Och95 for which McKPal95 estimated an LQRE.

[9]:
```
data = g.mixed_strategy_profile([[128*0.527, 128*(1-0.527)], [128*0.366, 128*(1-0.366)]])
fit = gbt.qre.logit_estimate(data)
type(fit)
```

[9]: pygambit.qre.LogitQREMixedStrategyFitResult

The returned `LogitQREMixedStrategyFitResult` object contains the results of the estimation. The results replicate those reported in McKPal95, including the estimated value of lambda, the QRE profile probabilities, and the log-likelihood.

Because `data` contains the empirical counts of play, and not just frequencies, the resulting log-likelihood is correct for use in likelihoood-ratio tests. [1]

[10]:
```
print(fit.lam)
print(fit.profile)
print(fit.log_like)
```

```
1.8456097536855864
[[0.615651314427859, 0.3843486855721409], [0.3832909400456291, 0.616709059954371]]
-174.76453191087444
```

All of the functions above also support working with the agent LQRE of McKPal98. Agent QRE are computed as the default behavior whenever the game has a extensive (tree) representation.

For `logit_solve`, `logit_solve_branch`, and `logit_solve_lambda`, this can be overriden by passing `use_strategic=True`; this will compute LQRE using the reduced strategy set of the game instead.

Likewise, `logit_estimate` will perform estimation using agent LQRE if the data passed are a `MixedBehaviorProfile`, and will return a `LogitQREMixedBehaviorFitResult` object.

Footnotes:

The log-likelihoods quoted in McKPal95 are exactly a factor of 10 larger than those obtained by replicating the calculation.

3.3 Interoperability tutorials

These tutorials demonstrate how to use PyGambit alongside other game-theoretic software packages. These tutorials assume you have read the new user tutorials and are familiar with the PyGambit API, however they do not assume prior knowledge of the other software packages or an advanced understanding of Game Theory:

3.3.1 Using Gambit with OpenSpiel

This tutorial demonstrates the interoperability of the Gambit and OpenSpiel Python packages for game-theoretic analysis.

Gambit provides a range of methods to compute exact and close approximations of equilibria for games. OpenSpiel provides a variety of iterative multi-agent learning algorithms, which may or may not converge to equilibria.

Another key distinction is that the PyGambit API allows the user a simple way to define custom games (see tutorials 1-3). This is also possible in OpenSpiel for normal-form games, and you can load `.efg` files created from Gambit for the extensive-form, however some of the key functionality for iterated learning of strategies is only available for games from the built-in library (see the OpenSpiel documentation).

This tutorial demonstrates:

1. Transferring examples of normal (strategic) form and extensive-form games between OpenSpiel and Gambit
2. Simulating evolutionary dynamics of populations of strategies in OpenSpiel for normal-form games
3. Training agents using self-play of extensive-form games in OpenSpiel to create strategies
4. Comparing the strategies from OpenSpiel against equilibrium strategies computed with Gambit

Note: The OpenSpiel code was adapted from the introductory tutorial for the OpenSpiel API on colab here.

[1]:
```
from io import StringIO

import matplotlib.pyplot as plt
import numpy as np
import pyspiel
from open_spiel.python import rl_environment
from open_spiel.python.algorithms import tabular_qlearner
from open_spiel.python.algorithms.gambit import export_gambit
from open_spiel.python.egt import dynamics
from open_spiel.python.egt.utils import game_payoffs_array

import pygambit as gbt
```

OpenSpiel game library

OpenSpiel has a large selection of games available in its library. Many of these will not be amenable to equilibrium computation with Gambit, due to their size. For the purposes of this tutorial, we'll pick some of the smallest games from the list below.

[2]:
```
print(pyspiel.registered_names())
```

```
['2048', 'add_noise', 'amazons', 'antichess', 'backgammon', 'bargaining', 'battleship',
↪'blackjack', 'blotto', 'breakthrough', 'bridge', 'bridge_uncontested_bidding', 'cached_
↪tree', 'catch', 'checkers', 'chess', 'cliff_walking', 'clobber', 'coin_game', 'colored_
↪trails', 'connect_four', 'coop_box_pushing', 'coop_to_1p', 'coordinated_mp', 'crazy_
↪eights', 'crazyhouse', 'cribbage', 'cursor_go', 'dark_chess', 'dark_hex', 'dark_hex_ir
↪', 'deep_sea', 'dots_and_boxes', 'dou_dizhu', 'efg_game', 'einstein_wurfelt_nicht',
```

(continues on next page)

(continued from previous page)

```
↪'euchre', 'first_sealed_auction', 'gin_rummy', 'go', 'gomoku', 'goofspiel', 'hanabi',
↪'havannah', 'hearts', 'hex', 'hive', 'kriegspiel', 'kuhn_poker', 'laser_tag', 'latent_
↪ttt', 'leduc_poker', 'lewis_signaling', 'liars_dice', 'liars_dice_ir', 'lines_of_action
↪', 'maedn', 'mancala', 'markov_soccer', 'matching_pennies_3p', 'matrix_bos', 'matrix_
↪brps', 'matrix_cd', 'matrix_coordination', 'matrix_mp', 'matrix_pd', 'matrix_rps',
↪'matrix_rpsw', 'matrix_sh', 'matrix_shapleys_game', 'mfg_crowd_modelling', 'mfg_crowd_
↪modelling_2d', 'mfg_dynamic_routing', 'mfg_garnet', 'misere', 'mnk', 'morpion_solitaire
↪', 'negotiation', 'nfg_game', 'nim', 'nine_mens_morris', 'normal_form_extensive_game',
↪'oh_hell', 'oshi_zumo', 'othello', 'oware', 'pathfinding', 'pentago', 'phantom_go',
↪'phantom_ttt', 'phantom_ttt_ir', 'pig', 'quoridor', 'rbc', 'repeated_game', 'repeated_
↪leduc_poker', 'repeated_poker', 'restricted_nash_response', 'sheriff', 'skat',
↪'solitaire', 'spades', 'start_at', 'stones_and_gems', 'tarok', 'tic_tac_toe', 'tiny_
↪bridge_2p', 'tiny_bridge_4p', 'tiny_hanabi', 'trade_comm', 'turn_based_simultaneous_
↪game', 'twixt', 'ultimate_tic_tac_toe', 'universal_poker', 'y', 'yacht', 'zerosum']
```

Normal-form games from the OpenSpiel library

Let's start with the simple normal-form game of rock-paper-scissors, in which the payoffs can be represented by a 3x3 matrix.

Load matrix rock-paper-scissors from OpenSpiel:

[3]:
```
ops_matrix_rps_game = pyspiel.load_game("matrix_rps")
```

In order to simulate a playthrough of the game, you can first initialise a game state:

[4]:
```
state = ops_matrix_rps_game.new_initial_state()
state
```

[4]:
```
Terminal? false
Row actions: Rock Paper Scissors
Col actions: Rock Paper Scissors
Utility matrix:
0,0 -1,1 1,-1
1,-1 0,0 -1,1
-1,1 1,-1 0,0
```

The possible actions for both players (player 0 and player 1) are Rock, Paper and Scissors, but these are not labelled and must be accessed via integer indices:

[5]:
```
print(state.legal_actions(0)) # Player 0 (row) actions
print(state.legal_actions(1)) # Player 1 (column) actions
```

```
[0, 1, 2]
[0, 1, 2]
```

Since Rock-paper-scissors is a 1-step simultaneous-move normal-form game, we'll apply a list of player actions in one step to reach the terminal state.

Let's simulate player 0 playing Rock (0) and player 1 playing Paper (1):

[6]:
```
state.apply_actions([0, 1])
state
```

[6]:
```
Terminal? true
History: 0, 1
Returns: -1,1
Row actions:
Col actions:
Utility matrix:
0,0 -1,1 1,-1
1,-1 0,0 -1,1
-1,1 1,-1 0,0
```

OpenSpiel can generate an NFG representation of the game loadable in Gambit:

[7]:
```
nfg_matrix_rps_game = pyspiel.game_to_nfg_string(ops_matrix_rps_game)
nfg_matrix_rps_game
```

[7]:
```
'NFG 1 R "OpenSpiel export of matrix_rps()"\n{ "Player 0" "Player 1" } { 3 3 }\n\n0 0\n1␣
↪-1\n-1 1\n-1 1\n0 0\n1 -1\n1 -1\n-1 1\n0 0\n'
```

Now let's load the NFG in Gambit. Since Gambit's `read_nfg` function expects a file like object, we'll convert the string with `io.StringIO`. We can also add labels for the actions to make the output more interpretable:

[8]:
```
gbt_matrix_rps_game = gbt.read_nfg(StringIO(nfg_matrix_rps_game))

gbt_matrix_rps_game.title = "Rock-Paper-Scissors"

for player in gbt_matrix_rps_game.players:
    player.strategies[0].label = "Rock"
    player.strategies[1].label = "Paper"
    player.strategies[2].label = "Scissors"

gbt_matrix_rps_game
```

[8]:
```
Game(title='Rock-Paper-Scissors')
```

The unique equilibrium mixed strategy profile for both players is to choose rock, paper, and scissors with equal probability:

[9]:
```
gbt.nash.lcp_solve(gbt_matrix_rps_game).equilibria[0]
```

[9]: $\left[\left[\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right], \left[\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right]\right]$

We can use OpenSpiel's dynamics module to demonstrate evolutionary game theory dynamics, or "replicator dynamics", which models how a mixed strategy profile evolves over time based on how the strategies (e.g., choice of actions A, B, C with probabilities X, Y, Z) perform against one another.

Let's start with an initial profile that is not at equilibrium, but weighted towards scissors with proportions: 30% Rock, 30% Paper, 40% Scissors:

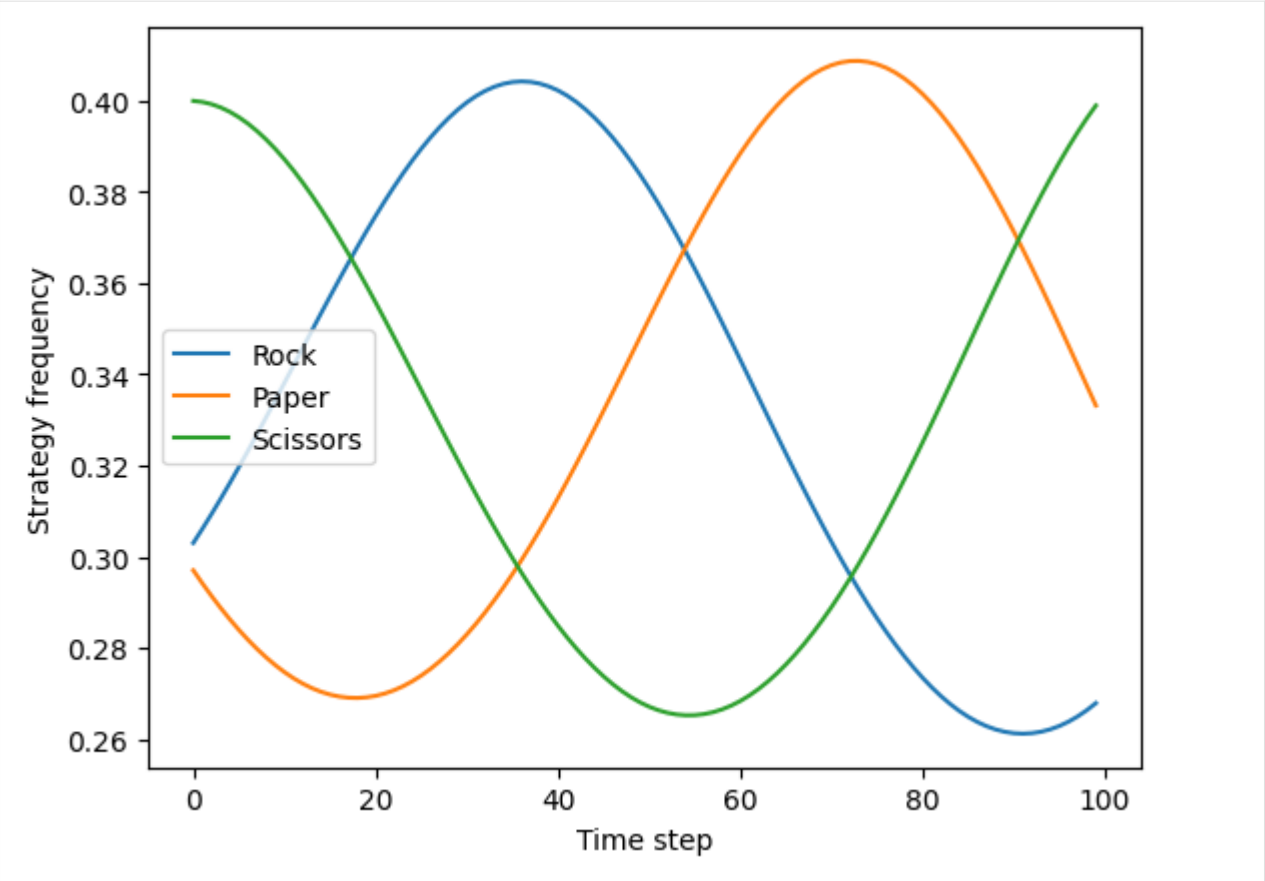
[10]:
```
matrix_rps_payoffs = game_payoffs_array(ops_matrix_rps_game)
dyn = dynamics.SinglePopulationDynamics(matrix_rps_payoffs, dynamics.replicator)
x = np.array([0.3, 0.3, 0.4])
dyn(x)
```

[10]:
```
array([ 3.00000000e-02, -3.00000000e-02, -1.66533454e-19])
```

`dyn(x)` calculates the rate of change (derivative) for each strategy in the current profile and returns how fast each strategy's frequency is changing.

In replicator dynamics, a pure strategy that performs well against others will increase in frequency, while strategies performing worse will decrease. In our rock-paper-scissors example, the performance of each pure strategy (action) depends on the probability it is assigned in the mixed strategy profile. At the start, whilst there are more players choosing scissors as their action, then rock will perform well and increase in frequency (be more likely to get played in subsequent rounds), while paper will perform poorly and decrease in frequency. We can plot how the frequency of each strategy changes over time:

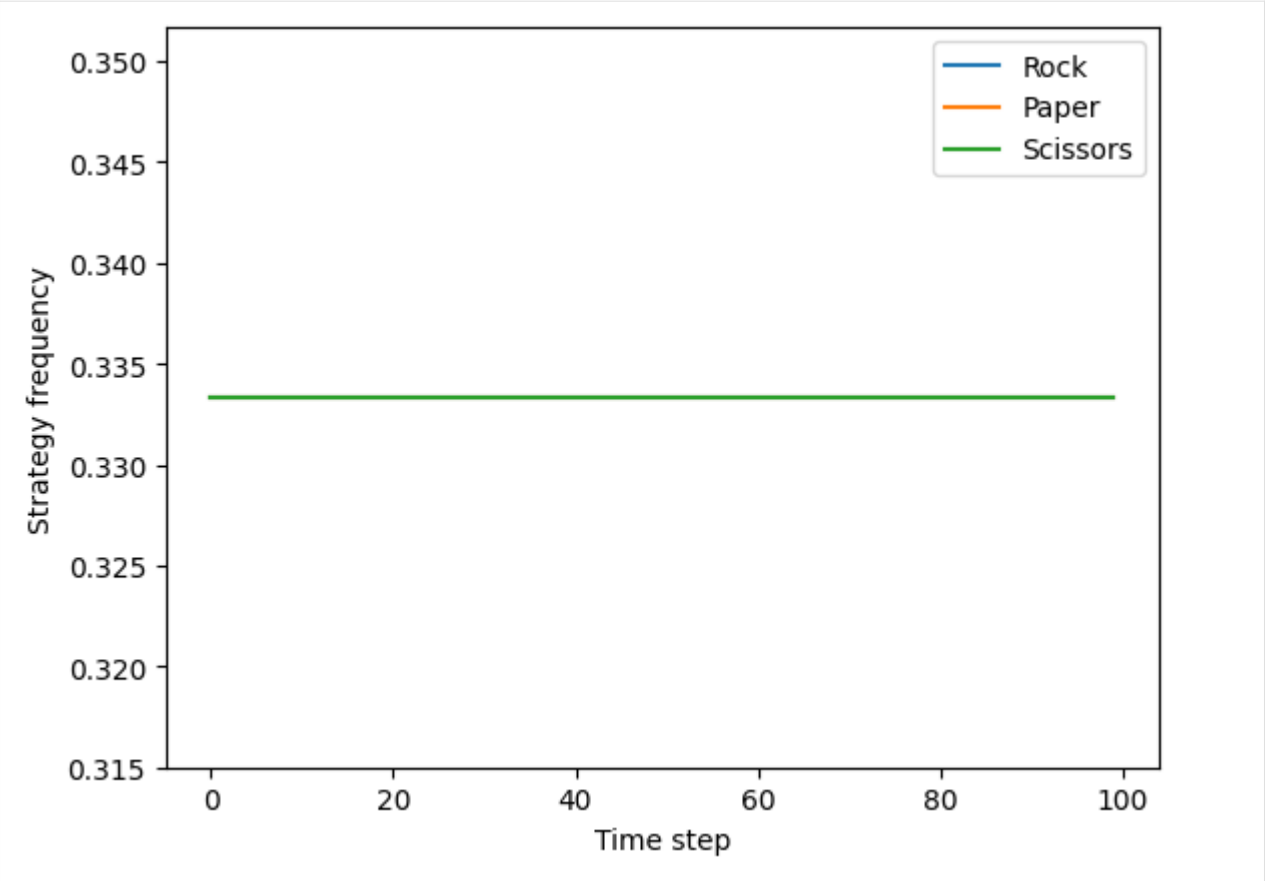
[11]:
```
def plot_rps_dynamics(proportions, steps=100, alpha=0.1, plot_average_strategy=False):
    x = np.array(proportions)
    rock_proportions = [x[0]]
    paper_proportions = [x[1]]
    scissors_proportions = [x[2]]
    y = []
    for _ in range(steps):
        x += alpha * dyn(x)
        rock_proportions.append(x[0])
        paper_proportions.append(x[1])
        scissors_proportions.append(x[2])
        if plot_average_strategy:
            y.append([np.mean(rock_proportions),
                      np.mean(paper_proportions),
                      np.mean(scissors_proportions)
                      ])
        else:
            y.append(x.copy())
    y = np.array(y)

    plt.plot(y[:, 0], label="Rock")
    plt.plot(y[:, 1], label="Paper")
    plt.plot(y[:, 2], label="Scissors")
    plt.xlabel("Time step")
    if plot_average_strategy:
        plt.ylabel("Strategy frequency average up to time step")
    else:
        plt.ylabel("Strategy frequency")
    plt.legend()
    plt.show()

plot_rps_dynamics([0.3, 0.3, 0.4])
```



Through the dynamics, we can see that the population proportions oscillate around the equilibrium point (1/3, 1/3, 1/3) without converging to it, because the best strategy depends on the likelihood of the opponents' actions, as defined by the current action probabilities.

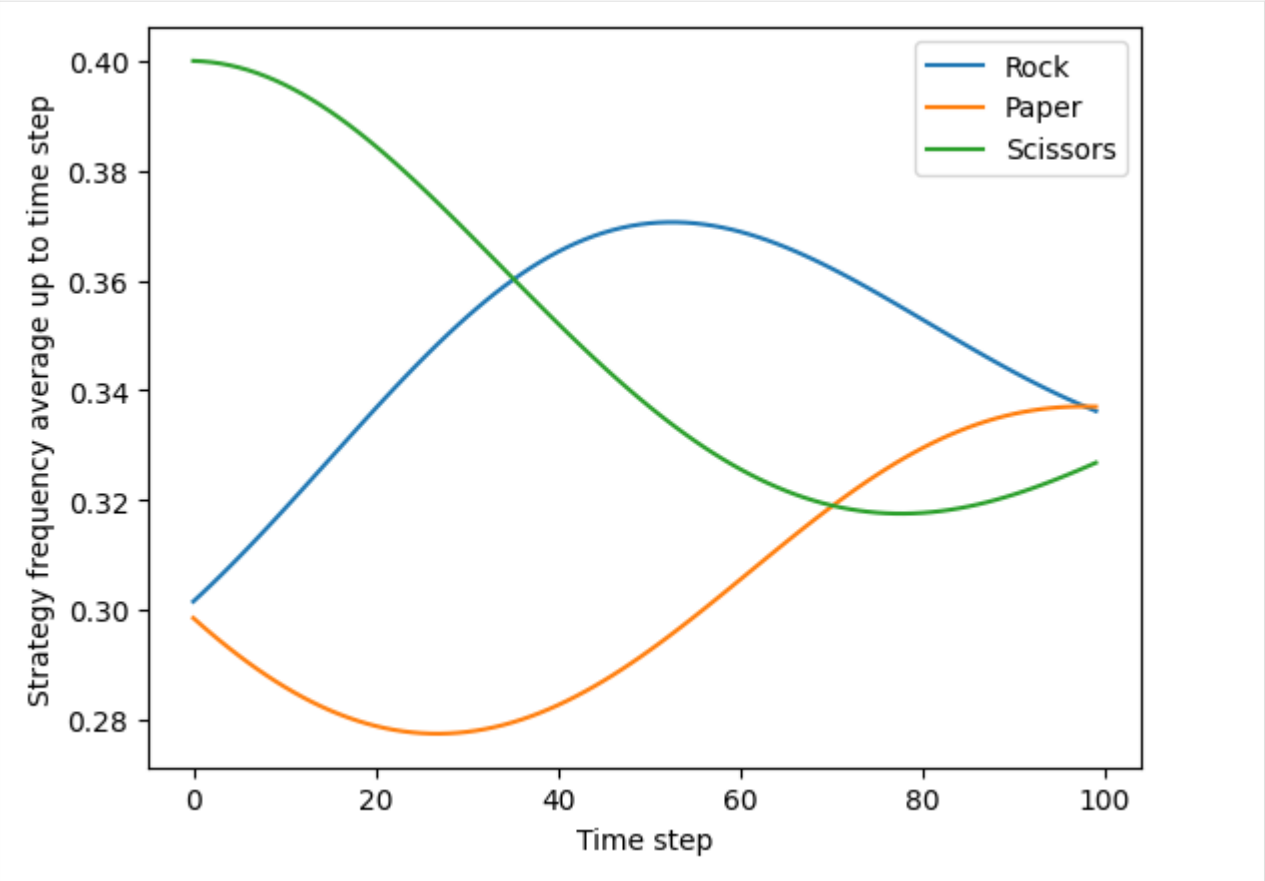
However, if we start with the initial population already at the equilibrium mixed strategy profile computed by Gambit (each action is chosen exactly 1/3 of the time), the strategy frequencies will remain constant over time (at the equilibrium point):

[12]:
```
plot_rps_dynamics([1/3, 1/3, 1/3])
```



When starting from an unbalanced initial mixed strategy profile, the strategy frequencies will oscillate around the equilibrium point without converging to it. However, if we plot the average strategy frequencies over time, we can see that this begins to converge to the equilibrium point:

[13]:
```
plot_rps_dynamics([0.3, 0.3, 0.4], plot_average_strategy=True)
```



Normal-form games created with Gambit

You can also set up a normal-form game in Gambit and export it to OpenSpiel. Here we demonstrate this with the simple Prisoner's Dilemma game:

[14]:
```
player1_payoffs = np.array([[-1, -3], [0, -2]])
player2_payoffs = np.transpose(player1_payoffs)

gbt_prisoners_dilemma_game = gbt.Game.from_arrays(
    player1_payoffs,
    player2_payoffs,
    title="Prisoner's Dilemma"
)
gbt_prisoners_dilemma_game
```

[14]:
```
Game(title='Prisoner's Dilemma')
```

[15]:
```
gbt.nash.lcp_solve(gbt_prisoners_dilemma_game).equilibria[0]
```

[15]: $[[0, 1], [0, 1]]$

As expected, Gambit computes the unique equilibrium strategy for both players as choosing cooperate with probability 0 and defect with probability 1.

To re-create the game in OpenSpiel we extract the player payoffs to NumPy arrays, which are then used to create a matrix game in OpenSpiel:

[16]:
```
p1_payoffs, p2_payoffs = gbt_prisoners_dilemma_game.to_arrays(dtype=float)
ops_prisoners_dilemma_game = pyspiel.create_matrix_game(
    gbt_prisoners_dilemma_game.title,
    "Classic Prisoner's Dilemma",  # description
    [strategy.label for strategy in gbt_prisoners_dilemma_game.players[0].strategies],
    [strategy.label for strategy in gbt_prisoners_dilemma_game.players[1].strategies],
    p1_payoffs,
    p2_payoffs
)
```

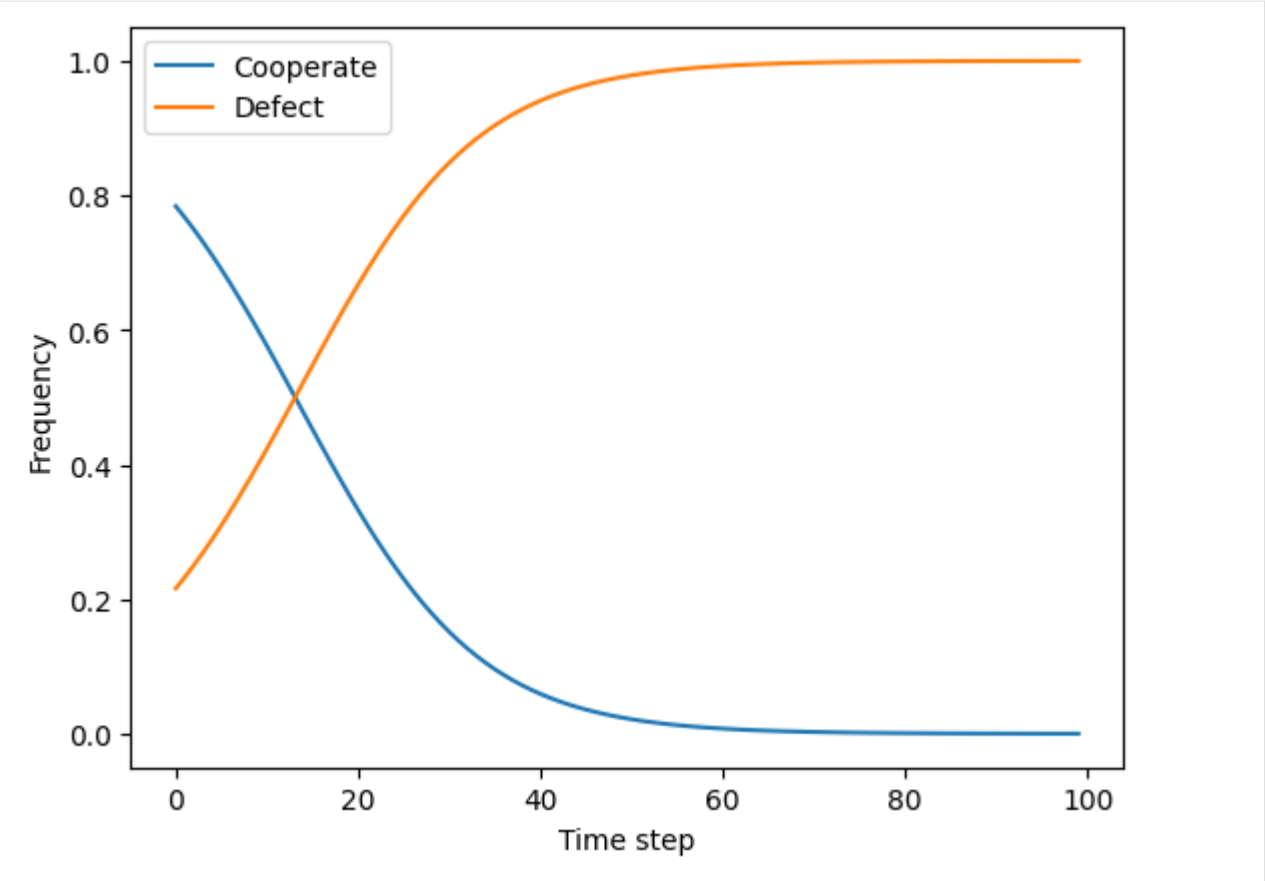
Like rock-paper-scissors, the Prisoner's Dilemma is a 1-step simultaneous-move normal-form game; we'll apply a list of player actions in one step to reach the terminal state. Let's have both player choose to defect (1):

[17]:
```
state = ops_prisoners_dilemma_game.new_initial_state()
state.apply_actions([1, 1])
state
```

[17]:
```
Terminal? true
History: 1, 1
Returns: -2,-2
Row actions:
Col actions:
Utility matrix:
-1,-1 -3,0
0,-3 -2,-2
```

Unlike in rock-paper-scissors, the Prisoner's Dilemma has a dominant strategy equilibrium, in which both players defect. Using evolutionary dynamics, we can see that a population starting with a mix of cooperators and defectors will evolve towards all defectors over time:

[18]:
```
matrix_pd_payoffs = game_payoffs_array(ops_prisoners_dilemma_game)
pd_dyn = dynamics.SinglePopulationDynamics(matrix_pd_payoffs, dynamics.replicator)

def plot_pd_dynamics(proportions, steps=100, alpha=0.1):
    x = np.array(proportions)
    y = []
    for _ in range(steps):
        x += alpha * pd_dyn(x)
        y.append(x.copy())
    y = np.array(y)
    plt.plot(y[:, 0], label="Cooperate")
    plt.plot(y[:, 1], label="Defect")
    plt.xlabel("Time step")
    plt.ylabel("Frequency")
    plt.legend()
    plt.show()

plot_pd_dynamics([0.8, 0.2])
```



Extensive-form games from the OpenSpiel library

For extensive-form games, OpenSpiel can export to the EFG format used by Gambit. Here we demonstrate this with **Tiny Hanabi**, loaded from the OpenSpiel game library:

[19]:
```
ops_hanabi_game = pyspiel.load_game("tiny_hanabi")
efg_hanabi_game = export_gambit(ops_hanabi_game)
efg_hanabi_game
```

[19]:
```
'EFG 2 R "tiny_hanabi()" { "Pl0" "Pl1" } \nc "" 1 "" { "d0" 0.50000000000000000 "d1" 0.
↪50000000000000000  } 0\n c "p0:d0" 2 "" { "d0" 0.50000000000000000 "d1" 0.
↪50000000000000000  } 0\n  p "" 1 1 "" { "p0a0" "p0a1" "p0a2"  } 0\n   p "" 2 1 "" {
↪"p1a0" "p1a1" "p1a2"  } 0\n    t "" 1 "" { 10.0 10.0 }\n    t "" 2 "" { 0.0 0.0 }\n   ␣
↪t "" 3 "" { 0.0 0.0 }\n   p "" 2 2 "" { "p1a0" "p1a1" "p1a2"  } 0\n    t "" 4 "" { 4.0␣
↪4.0 }\n    t "" 5 "" { 8.0 8.0 }\n    t "" 6 "" { 4.0 4.0 }\n   p "" 2 3 "" { "p1a0"
↪"p1a1" "p1a2"  } 0\n    t "" 7 "" { 10.0 10.0 }\n    t "" 8 "" { 0.0 0.0 }\n    t "" 9
↪"" { 0.0 0.0 }\n  p "" 1 1 "" { "p0a0" "p0a1" "p0a2"  } 0\n   p "" 2 4 "" { "p1a0"
↪"p1a1" "p1a2"  } 0\n    t "" 10 "" { 0.0 0.0 }\n    t "" 11 "" { 0.0 0.0 }\n    t ""␣
↪12 "" { 10.0 10.0 }\n   p "" 2 5 "" { "p1a0" "p1a1" "p1a2"  } 0\n    t "" 13 "" { 4.0␣
↪4.0 }\n    t "" 14 "" { 8.0 8.0 }\n    t "" 15 "" { 4.0 4.0 }\n   p "" 2 6 "" { "p1a0"
↪"p1a1" "p1a2"  } 0\n    t "" 16 "" { 0.0 0.0 }\n    t "" 17 "" { 0.0 0.0 }\n    t ""␣
↪18 "" { 10.0 10.0 }\n c "p0:d1" 3 "" { "d0" 0.50000000000000000 "d1" 0.50000000000000000␣
↪ } 0\n  p "" 1 2 "" { "p0a0" "p0a1" "p0a2"  } 0\n   p "" 2 1 "" { "p1a0" "p1a1" "p1a2"␣
↪ } 0\n    t "" 19 "" { 0.0 0.0 }\n    t "" 20 "" { 0.0 0.0 }\n    t "" 21 "" { 10.0 10.
↪0 }\n   p "" 2 2 "" { "p1a0" "p1a1" "p1a2"  } 0\n    t "" 22 "" { 4.0 4.0 }\n    t ""␣
```

(continues on next page)

(continued from previous page)

```
↪23 "" { 8.0 8.0 }\n    t "" 24 "" { 4.0 4.0 }\n   p "" 2 3 "" { "p1a0" "p1a1" "p1a2"  }
↪ 0\n    t "" 25 "" { 0.0 0.0 }\n    t "" 26 "" { 0.0 0.0 }\n    t "" 27 "" { 0.0 0.0 }\
↪n  p "" 1 2 "" { "p0a0" "p0a1" "p0a2"  } 0\n   p "" 2 4 "" { "p1a0" "p1a1" "p1a2"  } 0\
↪n    t "" 28 "" { 10.0 10.0 }\n    t "" 29 "" { 0.0 0.0 }\n    t "" 30 "" { 0.0 0.0 }\
↪n   p "" 2 5 "" { "p1a0" "p1a1" "p1a2"  } 0\n    t "" 31 "" { 4.0 4.0 }\n    t "" 32 "
↪" { 8.0 8.0 }\n    t "" 33 "" { 4.0 4.0 }\n   p "" 2 6 "" { "p1a0" "p1a1" "p1a2"  } 0\
↪n    t "" 34 "" { 10.0 10.0 }\n    t "" 35 "" { 0.0 0.0 }\n    t "" 36 "" { 0.0 0.0 }\n
↪'
```

Now let's load the EFG in Gambit. We can then compute equilibria strategies for the players as usual:

[20]:
```
gbt_hanabi_game = gbt.read_efg(StringIO(efg_hanabi_game))
eqm = gbt.nash.lcp_solve(gbt_hanabi_game).equilibria[0]
```

[21]:
```
from draw_tree import draw_tree

draw_tree(
    gbt_hanabi_game,
    color_scheme="gambit",
    edge_thickness=2,
    action_label_position=0.8,
    shared_terminal_depth=True
)
```

[21]:

We can look at player 0's equilibrium strategy:

[22]:
```
eqm["Pl0"]
```

[22]: $[[0, 0, 1], [0, 1, 0]]$

...and use Gambit to explore what those numbers actually mean for player 0:

[23]:
```
for infoset, mixed_action in eqm["Pl0"].mixed_actions():
    print(
        f"At information set {infoset.number}, "
        f"Player 0 plays action 0 with probability: {mixed_action['p0a0']}"
        f" and action 1 with probability: {mixed_action['p0a1']}"
        f" and action 2 with probability: {mixed_action['p0a2']}"
    )
```

```
At information set 0, Player 0 plays action 0 with probability: 0 and action 1 with␣
↪probability: 0 and action 2 with probability: 1
At information set 1, Player 0 plays action 0 with probability: 0 and action 1 with␣
↪probability: 1 and action 2 with probability: 0
```

For player 1, we can do the same:

[24]:
```
eqm["Pl1"]
```

[24]: $[[0, 0, 1], [0, 1, 0], [1, 0, 0], [0, 0, 1], [0, 1, 0], [0, 0, 1]]$

[25]:
```
for infoset, mixed_action in eqm["Pl1"].mixed_actions():
    print(
```

(continues on next page)

(continued from previous page)

```
        f"At information set {infoset.number}, "
        f"Player 1 plays action 0 with probability: {mixed_action['p1a0']}"
        f" and action 1 with probability: {mixed_action['p1a1']}"
        f" and action 2 with probability: {mixed_action['p1a2']}"
    )
```

```
At information set 0, Player 1 plays action 0 with probability: 0 and action 1 with␣
↪probability: 0 and action 2 with probability: 1
At information set 1, Player 1 plays action 0 with probability: 0 and action 1 with␣
↪probability: 1 and action 2 with probability: 0
At information set 2, Player 1 plays action 0 with probability: 1 and action 1 with␣
↪probability: 0 and action 2 with probability: 0
At information set 3, Player 1 plays action 0 with probability: 0 and action 1 with␣
↪probability: 0 and action 2 with probability: 1
At information set 4, Player 1 plays action 0 with probability: 0 and action 1 with␣
↪probability: 1 and action 2 with probability: 0
At information set 5, Player 1 plays action 0 with probability: 0 and action 1 with␣
↪probability: 0 and action 2 with probability: 1
```

Let's now train 2 agents using independent Q-learning on Tiny Hanabi, and play them against eachother.

We can compare the learned strategies played to the equilibrium strategies computed by Gambit.

First let's open the RL environment for Tiny Hanabi and create the agents, one for each player (2 players in this case):

[26]:
```
# Create the environment
env = rl_environment.Environment("tiny_hanabi")
num_players = env.num_players
num_actions = env.action_spec()["num_actions"]

# Create the agents
agents = [
    tabular_qlearner.QLearner(player_id=idx, num_actions=num_actions)
    for idx in range(num_players)
]
```

Now we can train the Q-learning agents in self-play:

[27]:
```
for cur_episode in range(30000):
  if cur_episode % 10000 == 0:
    print(f"Episodes: {cur_episode}")

  time_step = env.reset()
  while not time_step.last():
    player_id = time_step.observations["current_player"]
    agent_output = agents[player_id].step(time_step)
    time_step = env.step([agent_output.action])

  # Episode is over, step all agents with final info state.
  for agent in agents:
    agent.step(time_step)

print(f"Episodes: {cur_episode+1}")
```

```
Episodes: 0
Episodes: 10000
Episodes: 20000
Episodes: 30000
```

Let's check out the strategies our agents have learned by playing them against eachother again, this time in evaluation mode (setting `is_evaluation=True`):

[28]:
```
time_step = env.reset()

while not time_step.last():
  print("")
  print(env.get_state)

  player_id = time_step.observations["current_player"]
  agent_output = agents[player_id].step(time_step, is_evaluation=True)
  print(f"Agent {player_id} chooses {env.get_state.action_to_string(agent_output.action)}
↪")
  time_step = env.step([agent_output.action])

print("")
print(env.get_state)
print(f"Rewards: {time_step.rewards}")
```

```

p0:d0 p1:d1
Agent 0 chooses p0a1

p0:d0 p1:d1 p0:a1
Agent 1 chooses p1a1

p0:d0 p1:d1 p0:a1 p1:a1
Rewards: [8.0, 8.0]
```

Are the learned strategies chosen by p0 and p1 consistent with an equilibrium computed by Gambit?

When I ran the above I got the final game state `p0:d0 p1:d0 p0:a2 p1:a0` with payoffs `[10.0, 10.0]`. This is consistent with the equilibrium computed by Gambit:

- The node `p0:d0 p1:d0` is part of player 0's information set 0.
- p0 picks a2 which matches the first equilibrium strategy in `eqm['Pl0']` where action `p0a2` is played with probability 1.0.
- This puts player 1 in their information set 2, and player 1 picks action 0, which is consistent with `eqm['Pl1']` where action `p1a0` is played with probability 1.0.

Extensive-form games created with Gambit

It's also possible to create an extensive-form game in Gambit and export it to OpenSpiel. Here we demonstrate this with the one-card poker game introduced in tutorial 3:

[29]:
```
gbt_one_card_poker = gbt.Game.new_tree(
    players=["Alice", "Bob"],
    title="Stripped-Down Poker: a simple game of one-card poker from Reiley et al (2008).
↪"
```

(continues on next page)

(continued from previous page)

```
)

gbt_one_card_poker.append_move(
    gbt_one_card_poker.root,
    player=gbt_one_card_poker.players.chance,
    actions=["King", "Queen"]  # By default, chance actions have equal probabilities
)

for node in gbt_one_card_poker.root.children:
    gbt_one_card_poker.append_move(
        node,
        player="Alice",
        actions=["Bet", "Fold"]
    )

gbt_one_card_poker.append_move(
    [
        gbt_one_card_poker.root.children["King"].children["Bet"],
        gbt_one_card_poker.root.children["Queen"].children["Bet"]
    ],
    player="Bob",
    actions=["Call", "Fold"]
)

win_big = gbt_one_card_poker.add_outcome([2, -2], label="Win Big")
win = gbt_one_card_poker.add_outcome([1, -1], label="Win")
lose_big = gbt_one_card_poker.add_outcome([-2, 2], label="Lose Big")
lose = gbt_one_card_poker.add_outcome([-1, 1], label="Lose")

# Alice folds, Bob wins small
gbt_one_card_poker.set_outcome(
    gbt_one_card_poker.root.children["King"].children["Fold"],
    lose
)
gbt_one_card_poker.set_outcome(
    gbt_one_card_poker.root.children["Queen"].children["Fold"],
    lose
)

# Bob sees Alice Bet and calls, correctly believing she is bluffing, Bob wins big
gbt_one_card_poker.set_outcome(
    gbt_one_card_poker.root.children["Queen"].children["Bet"].children["Call"],
    lose_big
)

# Bob sees Alice Bet and calls, incorrectly believing she is bluffing, Alice wins big
gbt_one_card_poker.set_outcome(
    gbt_one_card_poker.root.children["King"].children["Bet"].children["Call"],
    win_big
)

# Bob does not call Alice's Bet, Alice wins small
```

(continues on next page)

(continued from previous page)

```
gbt_one_card_poker.set_outcome(
    gbt_one_card_poker.root.children["King"].children["Bet"].children["Fold"],
    win
)
gbt_one_card_poker.set_outcome(
    gbt_one_card_poker.root.children["Queen"].children["Bet"].children["Fold"],
    win
)
```

[30]:
```
draw_tree(gbt_one_card_poker, color_scheme="gambit")
```

[30]:

Create the game in OpenSpiel:

[31]:
```
ops_one_card_poker = pyspiel.load_efg_game(gbt_one_card_poker.to_efg())
ops_one_card_poker
```

[31]:
```
efg_game()
```

Games loaded from EFG in OpenSpiel do not take advantage of the full functionality of the package, for example, it is not possible to carry out training with RL algorithms on these games, as in the example above with Tiny Hanabi. The OpenSpiel documentation explains how to submit new games to the library if you wish to add your own games.

We can however use the state representation to simulate a playthrough of the game:

[32]:
```
players = {0: "Alice", 1: "Bob", -1: "Chance"}

# Create an initial game state, then play through to completion
state = ops_one_card_poker.new_initial_state()
while not state.is_terminal():

  # Store legal actions of current player in a dict
  legal_actions = {}
  for action in state.legal_actions():
    legal_actions[action] = state.action_to_string(state.current_player(), action)

  # If player is chance, choose an action according to probability
  if state.is_chance_node():
    outcomes_with_probs = state.chance_outcomes()
    action_list, prob_list = zip(*outcomes_with_probs, strict=True)
    action = np.random.choice(action_list, p=prob_list)
    print("Dealt card: ", legal_actions[action])
    state.apply_action(action)

  # Regular players pick a random legal action.
  else:
    action = np.random.choice(state.legal_actions())
    print(players[state.current_player()], " action: ", legal_actions[action])
    state.apply_action(action)
  print()

print("Alice receives: ", state.player_return(0), ", Bob receives: ", state.player_
↪return(1))
```

```
Dealt card:  King

Alice  action:  Bet

Bob  action:  Call

Alice receives:  2.0 , Bob receives:  -2.0
```

Run the code cell above to simulate different possible playthroughs of the game.

3.4 API documentation

3.4.1 API documentation

Representation of games

Game	A game, the fundamental unit of analysis in game theory.
Player	A player in a Game.
Outcome	An outcome in a Game.
Node	A node in a Game.
Infoset	An information set in a Game.
Action	A choice available at an Infoset in a Game.
Strategy	A plan of action for a Player in a Game.

pygambit.gambit.Game

class pygambit.gambit.**Game**

A game, the fundamental unit of analysis in game theory.

Games may be represented in extensive or strategic form.

Methods

add_action(infoset[, before])	Add an action at the information set *infoset*.
add_outcome([payoffs, label])	Add a new outcome to the game.
add_player([label])	Add a new player to the game.
add_strategy(player[, label])	Add a new strategy to the set of strategies for *player*.
append_infoset(nodes, infoset)	Add a move in information set *infoset* at terminal *nodes*.
append_move(nodes, player, actions)	Add a move for *player* at terminal *nodes*.
copy_tree(src, dest)	Copy the subtree rooted at the node *src* to the node *dest*.
delete_action(action)	Deletes *action* from its information set.
delete_outcome(outcome)	Delete an outcome from the game.
delete_parent(node)	Delete the parent node of *node*.
delete_strategy(strategy)	Delete *strategy* from the game.
delete_tree(node)	Truncate the game tree at *node*, deleting the subtree beneath it.
from_arrays(*arrays[, title])	Create a new Game with a strategic representation.
from_dict(payoffs[, title])	Create a new Game with a strategic representation.

continues on next page

Table 2 – continued from previous page

insert_infoset(node, infoset)	Insert a move in information set *infoset* prior to the node *node*.
insert_move(node, player, actions)	Insert a move for *player* prior to the node *node*, with *actions* actions.
leave_infoset(node)	Remove this node from its information set.
mixed_behavior_profile([data, rational])	Create a mixed behavior profile over the game.
mixed_strategy_profile([data, rational])	Create a mixed strategy profile over the game.
move_tree(src, dest)	Move the subtree rooted at 'src' to 'dest'.
new_table(dim[, title])	Create a new `Game` with a strategic representation.
new_tree([players, title])	Create a new `Game` consisting of a trivial game tree, with one node, which is both root and terminal.
random_behavior_profile([denom, gen])	Create a *MixedBehaviorProfile* on the game, with probabilities drawn from the uniform distribution over the set of mixed behavior profiles.
random_strategy_profile([denom, gen])	Create a *MixedStrategy* on the game, with probabilities drawn from the uniform distribution over the set of mixed strategy profiles.
reveal(infoset, player)	Reveals the move made at *infoset* to *player*.
set_chance_probs(infoset, probs)	Set the action probabilities at chance information set *infoset*.
set_infoset(node, infoset)	Place *node* in the information set *infoset*.
set_outcome(node, outcome)	Set *outcome* to be the outcome at *node*.
set_player(infoset, player)	Set the player at an information set.
sort_infosets()	Sort information sets into a standard order.
strategy_support_profile([strategies])	Create a new *StrategySupportProfile* on the game.
to_arrays(dtype)	Generate the payoff tables for players represented as numpy arrays.
to_efg([filepath_or_buffer])	Save the game to an .efg file or return its serialized representation
to_html([filepath_or_buffer])	Export the game to HTML format.
to_latex([filepath_or_buffer])	Export the game to LaTeX format.
to_nfg([filepath_or_buffer])	Save the game to a .nfg file or return its serialized representation

Attributes

actions	The set of actions available in the game.
contingencies	An iterator over the contingencies in the game.
description	Get or set the description of the game.
infosets	The set of information sets in the game.
is_const_sum	Whether the game is constant sum.
is_perfect_recall	Whether the game is perfect recall.
is_tree	Return whether a game has a tree-based representation.
max_payoff	The maximum payoff to any player in any play of the game.
min_payoff	The minimum payoff to any player in any play of the game.
nodes	The set of nodes in the game.
outcomes	The set of outcomes in the game.
players	The set of players in the game.

continues on next page

Table 3 – continued from previous page

root	The root node of the game.
strategies	The set of strategies in the game.
title	Get or set the title of the game.

pygambit.gambit.Player

class pygambit.gambit.**Player**

A player in a Game.

Methods

Attributes

actions	Returns the set of actions available to the player at some information set.
game	Gets the Game to which the player belongs.
infosets	Returns the set of information sets at which the player has the decision.
is_chance	Returns whether the player is the chance player.
label	Gets or sets the text label of the player.
max_payoff	Returns the largest payoff for the player in any play of the game.
min_payoff	Returns the smallest payoff for the player in any play of the game.
number	Returns the number of the player in its game.
strategies	Returns the set of strategies belonging to the player.

pygambit.gambit.Outcome

class pygambit.gambit.**Outcome**

An outcome in a Game.

Methods

Attributes

game	Returns the game with which this outcome is associated.
label	The text label associated with this outcome.
number	Returns the number of the outcome in the game.

pygambit.gambit.Node

class pygambit.gambit.**Node**

A node in a Game.

Methods

is_successor_of(node)	Returns whether this node is a successor of *node*.

Attributes

children	The set of children of this node.
game	Gets the Game to which the node belongs.
infoset	The information set to which this node belongs.
is_strategy_reachable	Returns whether this node is reachable by any pure strategy profile.
is_subgame_root	Returns whether the node is the root of a proper subgame.
is_terminal	Returns whether this is a terminal node of the game.
label	The text label associated with the node.
next_sibling	The node which is immediately after this one in its parent's children.
outcome	Returns the outcome attached to the node.
own_prior_action	The last action taken by the node's owner before reaching this node.
parent	The parent of this node.
player	The player who makes the decision at this node.
plays	Returns a list of all terminal *Node* objects consistent with it.
prior_action	The action which leads to this node.
prior_sibling	The node which is immediately before this one in its parent's children.

pygambit.gambit.Infoset

class pygambit.gambit.**Infoset**

An information set in a Game.

Methods

precedes(node)	Return whether this information set precedes *node* in the game tree.

Attributes

actions	The set of actions at the information set.

continues on next page

Table 11 – continued from previous page

game	The Game to which the information set belongs.
is_absent_minded	Whether the information set is absent-minded.
is_chance	Whether the information set belongs to the chance player.
label	Get or set the text label of the information set.
members	The set of nodes which are members of the information set.
number	Returns the number of the information set for its player.
own_prior_actions	The set of actions taken by the player immediately preceding the member nodes in the information set.
player	The player who has the move at this information set.
plays	Returns a list of all terminal *Node* objects consistent with it.

pygambit.gambit.Action

class pygambit.gambit.**Action**

A choice available at an Infoset in a Game.

Methods

precedes(node)	Returns whether *node* precedes this action in the extensive game.

Attributes

infoset	Get the information set to which the action belongs.
label	Get or set the text label of the action.
number	Returns the number of the action at its information set.
plays	Returns a list of all terminal *Node* objects consistent with it.
prob	Get the probability a chance action is played.

pygambit.gambit.Strategy

class pygambit.gambit.**Strategy**

A plan of action for a Player in a Game.

Methods

action(infoset)	Get the action prescribed by a strategy for a given information set.

Attributes

game	The game to which the strategy belongs.
label	Get or set the text label associated with the strategy.
number	The number of the strategy.
player	The player to which the strategy belongs.

Creating, reading, and writing games

read_gbt(filepath_or_buffer[, normalize_labels])	Construct a game from its serialised representation in a GBT file.
read_efg(filepath_or_buffer[, normalize_labels])	Construct a game from its serialised representation in an EFG file.
read_nfg(filepath_or_buffer[, normalize_labels])	Construct a game from its serialised representation in a NFG file.
read_agg(filepath_or_buffer[, normalize_labels])	Construct a game from its serialised representation in an AGG file.
Game.new_tree([players, title])	Create a new Game consisting of a trivial game tree, with one node, which is both root and terminal.
Game.new_table(dim[, title])	Create a new Game with a strategic representation.
Game.from_arrays(*arrays[, title])	Create a new Game with a strategic representation.
Game.to_arrays(dtype)	Generate the payoff tables for players represented as numpy arrays.
Game.from_dict(payoffs[, title])	Create a new Game with a strategic representation.
Game.to_efg([filepath_or_buffer])	Save the game to an .efg file or return its serialized representation
Game.to_nfg([filepath_or_buffer])	Save the game to a .nfg file or return its serialized representation
Game.to_html([filepath_or_buffer])	Export the game to HTML format.
Game.to_latex([filepath_or_buffer])	Export the game to LaTeX format.

pygambit.gambit.read_gbt

pygambit.gambit.**read_gbt**(*filepath_or_buffer: str | Path | IOBase*, *normalize_labels: bool = False*) → *Game*

Construct a game from its serialised representation in a GBT file.

Parameters

- **filepath_or_buffer** (*str, pathlib.Path or io.IOBase*) – The path to the file containing the game representation or file-like object
- **normalize_labels** (*bool (default False)*) – Ensure all labels are nonempty and unique within their scopes. This will be enforced in a future version of Gambit.

Returns

A game constructed from the representation in the file.

Return type

Game

Raises

- **IOError** – If the file cannot be opened or read
- **ValueError** – If the contents of the file are not a valid game representation.

See also

read_efg, *read_nfg*, *read_agg*

pygambit.gambit.read_efg

pygambit.gambit.**read_efg**(*filepath_or_buffer: str | Path | IOBase*, *normalize_labels: bool = False*) → *Game*

Construct a game from its serialised representation in an EFG file.

Parameters

- **filepath_or_buffer** (*str, pathlib.Path or io.IOBase*) – The path to the file containing the game representation or file-like object
- **normalize_labels** (*bool (default False)*) – Ensure all labels are nonempty and unique within their scopes. This will be enforced in a future version of Gambit.

Returns

A game constructed from the representation in the file.

Return type

Game

Raises

- **IOError** – If the file cannot be opened or read
- **ValueError** – If the contents of the file are not a valid game representation.

See also

read_gbt, *read_nfg*, *read_agg*

pygambit.gambit.read_nfg

pygambit.gambit.**read_nfg**(*filepath_or_buffer: str | Path | IOBase*, *normalize_labels: bool = False*) → *Game*

Construct a game from its serialised representation in a NFG file.

Parameters

- **filepath_or_buffer** (*str, pathlib.Path or io.IOBase*) – The path to the file containing the game representation or file-like object
- **normalize_labels** (*bool (default False)*) – Ensure all labels are nonempty and unique within their scopes. This will be enforced in a future version of Gambit.

Returns

A game constructed from the representation in the file.

Return type

Game

Raises

- **IOError** – If the file cannot be opened or read

- **ValueError** – If the contents of the file are not a valid game representation.

See also

read_gbt, *read_efg*, *read_agg*

pygambit.gambit.read_agg

pygambit.gambit.**read_agg**(*filepath_or_buffer: str | Path | IOBase*, *normalize_labels: bool = False*) → *Game*

Construct a game from its serialised representation in an AGG file.

Parameters

- **filepath_or_buffer** (*str, pathlib.Path or io.IOBase*) – The path to the file containing the game representation or file-like object
- **normalize_labels** (*bool (default False)*) – Ensure all labels are nonempty and unique within their scopes. This will be enforced in a future version of Gambit.

Returns

A game constructed from the representation in the file.

Return type

Game

Raises

- **IOError** – If the file cannot be opened or read
- **ValueError** – If the contents of the file are not a valid game representation.

See also

read_gbt, *read_efg*, *read_nfg*

pygambit.gambit.Game.new_tree

classmethod Game.**new_tree**(*players: list[str] | None = None*, *title: str = 'Untitled extensive game'*) → *Game*

Create a new Game consisting of a trivial game tree, with one node, which is both root and terminal.

Changed in version 16.1.0: Added the *players* and *title* parameters

Parameters

- **players** (*list of str, optional*) – A list of labels for the (strategic) players of the game. If *players* is not specified, the game initially has no players defined other than the chance player.
- **title** (*str, optional*) – The title of the game. If no title is specified, "Untitled extensive game" is used.

Returns

The newly-created extensive game.

Return type

Game

pygambit.gambit.Game.new_table

classmethod Game.**new_table**(*dim*, *title: str = 'Untitled strategic game'*) → *Game*

Create a new Game with a strategic representation.

Changed in version 16.1.0: Added the *title* parameter.

Parameters

- **dim** (*array-like*) – A list specifying the number of strategies for each player.
- **title** (*str, optional*) – The title of the game. If no title is specified, "Untitled strategic game" is used.

Returns
The newly-created strategic game.

Return type
Game

pygambit.gambit.Game.from_arrays

classmethod Game.**from_arrays**(**arrays*, *title: str = 'Untitled strategic game'*) → *Game*

Create a new Game with a strategic representation.

Each entry in *arrays* gives the payoff matrix for the corresponding player. The arrays must all have the same shape, and have the same number of dimensions as the total number of players.

Changed in version 16.1.0: Added the *title* parameter.

Parameters

- **arrays** (*array-like of array-like*) – The payoff matrices for the players.
- **title** (*str, optional*) – The title of the game. If no title is specified, "Untitled strategic game" is used.

Returns
The newly-created strategic game.

Return type
Game

See also

from_dict
Create strategic game and set player labels

to_array
Generate the payoff tables for players represented as numpy arrays

pygambit.gambit.Game.to_arrays

Game.**to_arrays**(*dtype: Type = <class 'pygambit.gambit.Rational'>*) → list[array]

Generate the payoff tables for players represented as numpy arrays.

Parameters
dtype (*type*) – The type to which payoff values will be converted and the resulting arrays will be of that dtype

Return type
list of np.array

See also

from_arrays
Create game from list-like of array-like

pygambit.gambit.Game.from_dict

classmethod Game.**from_dict**(*payoffs*, *title: str = 'Untitled strategic game'*) → *Game*

Create a new Game with a strategic representation.

Each entry in *payoffs* is a key-value pair giving the label and the payoff matrix for a player. The payoff matrices must all have the same shape, and have the same number of dimensions as the total number of players.

Parameters

- **payoffs** (*dict-like mapping str to array-like*) – The names and corresponding payoff matrices for the players.
- **title** (*str, optional*) – The title of the game. If no title is specified, "Untitled strategic game" is used.

Returns
The newly-created strategic game.

Return type
Game

See also

from_arrays
Create game from list-like of array-like

pygambit.gambit.Game.to_efg

Game.**to_efg**(*filepath_or_buffer: str | Path | IOBase | None = None*) → str | None

Save the game to an .efg file or return its serialized representation

Parameters
filepath_or_buffer (*str or Path or io.IOBase or None, default None*) – String, path object, or file-like object implementing a write() function. If None, the result is returned as a string.

Return type
String representation of the game or None if the game is saved to a file

See also

to_nfg, *to_html*, *to_latex*

pygambit.gambit.Game.to_nfg

Game.**to_nfg**(*filepath_or_buffer: str | Path | IOBase | None = None*) → str | None

Save the game to a .nfg file or return its serialized representation

Parameters

filepath_or_buffer (*str or Path or BufferedWriter or None, default None*) – String, path object, or file-like object implementing a write() function. If None, the result is returned as a string.

Return type

String representation of the game or None if the game is saved to a file

See also

to_efg, *to_html*, *to_latex*

pygambit.gambit.Game.to_html

Game.**to_html**(*filepath_or_buffer: str | Path | IOBase | None = None*) → str | None

Export the game to HTML format.

Generates a rendering of the strategic form of the game as a collection of HTML tables. The first player is the row chooser; the second player the column chooser. For games with more than two players, a collection of tables is generated, one for each possible strategy combination of players 3 and higher.

Parameters

filepath_or_buffer (*str or Path or BufferedWriter or None, default None*) – String, path object, or file-like object implementing a write() function. If None, the result is returned as a string.

Return type

String representation of the game or None if the game is exported to a file

See also

to_efg, *to_nfg*, *to_latex*

pygambit.gambit.Game.to_latex

Game.**to_latex**(*filepath_or_buffer: str | Path | IOBase | None = None*) → str | None

Export the game to LaTeX format.

Generates a rendering of the strategic form of the game in LaTeX, suitable for use with Martin Osborne's sgame style. The first player is the row chooser; the second player the column chooser. For games with more than two players, a collection of tables is generated, one for each possible strategy combination of players 3 and higher.

Parameters

filepath_or_buffer (*str or Path or BufferedWriter or None, default None*) – String, path object, or file-like object implementing a write() function. If None, the result is returned as a string.

Return type

String representation of the game or None if the game is exported to a file

See also

to_efg, *to_nfg*, *to_html*

Transforming game trees

Game.append_move(nodes, player, actions)	Add a move for *player* at terminal *nodes*.
Game.append_infoset(nodes, infoset)	Add a move in information set *infoset* at terminal *nodes*.
Game.insert_move(node, player, actions)	Insert a move for *player* prior to the node *node*, with *actions* actions.
Game.insert_infoset(node, infoset)	Insert a move in information set *infoset* prior to the node *node*.
Game.copy_tree(src, dest)	Copy the subtree rooted at the node *src* to the node *dest*.
Game.move_tree(src, dest)	Move the subtree rooted at 'src' to 'dest'.
Game.delete_parent(node)	Delete the parent node of *node*.
Game.delete_tree(node)	Truncate the game tree at *node*, deleting the subtree beneath it.

pygambit.gambit.Game.append_move

Game.**append_move**(*nodes:* Node | *Iterable[*Node | *str]*, *player:* Player | *str*, *actions: list[str]*) → None

Add a move for *player* at terminal *nodes*. All elements of *nodes* become part of a new information set, with actions labeled according to *actions*.

Raises

- **UndefinedOperationError** – If *nodes* are not all terminal, or *actions* is not a positive number.
- **MismatchError** – If an element from *nodes* is a *Node* from a different game, or *player* is a *Player* from a different game.
- **ValueError** – If *nodes* has duplicated elements, or is empty.

pygambit.gambit.Game.append_infoset

Game.**append_infoset**(*nodes:* Node | *Iterable[*Node | *str]*, *infoset:* Infoset | *str*) → None

Add a move in information set *infoset* at terminal *nodes*.

Raises

- **UndefinedOperationError** – If any element in *nodes* is not a terminal node.
- **MismatchError** – If an element in *nodes* is a *Node* from a different game, or *infoset* is an *Infoset* from a different game.
- **ValueError** – If *nodes* has duplicated elements, or is empty.

pygambit.gambit.Game.insert_move

Game.**insert_move**(*node:* Node | *str*, *player:* Player | *str*, *actions: int*) → None

Insert a move for *player* prior to the node *node*, with *actions* actions. *node* becomes the first child of the newly-inserted node.

Raises

- **UndefinedOperationError** – If *actions* is not a positive number.
- **MismatchError** – If *node* is a *Node* from a different game, or *player* is a *Player* from a different game.

pygambit.gambit.Game.insert_infoset

Game.**insert_infoset**(*node:* Node | *str*, *infoset:* Infoset | *str*) → None

Insert a move in information set *infoset* prior to the node *node*. *node* becomes the first child of the newly-inserted node.

Raises

MismatchError – If *node* is a *Node* from a different game, or *infoset* is an *Infoset* from a different game.

pygambit.gambit.Game.copy_tree

Game.**copy_tree**(*src:* Node | *str*, *dest:* Node | *str*) → None

Copy the subtree rooted at the node *src* to the node *dest*.

Each node in the subtree copied to follow *dest* is placed in the same information set as the corresponding node in the original subtree under *src*.

It is permitted for *dest* to be a descendant of *src*. The operation uses the subtree rooted at *src* as it is at the time the function is called, so no infinite recursion is triggered.

The outcome associated with *dest* is not changed by this operation.

Parameters

- **src** (Node *or* *str*) – The root of the source subtree to copy
- **dest** (Node *or* *str*) – The destination subtree to copy to. *dest* must be a terminal node.

Raises

- **MismatchError** – If *src* or *dest* is not a member of the same game as this node.
- **UndefinedOperationError** – If *dest* is not a terminal node.

pygambit.gambit.Game.move_tree

Game.**move_tree**(*src:* Node | *str*, *dest:* Node | *str*) → None

Move the subtree rooted at 'src' to 'dest'.

Parameters

- **src** (Node *or* *str*) – The root of the source subtree to move
- **dest** (Node *or* *str*) – The destination subtree to move to. *dest* must be a terminal node.

Raises

- **MismatchError** – If *src* or *dest* is not a member of the same game as this node.
- **UndefinedOperationError** – If *dest* is not a terminal node, or *dest* is a successor of *src*.

pygambit.gambit.Game.delete_parent

Game.**delete_parent**(*node:* Node | *str*) → None

Delete the parent node of *node*. *node* replaces its parent in the tree. All other subtrees rooted at *node*'s parent are deleted.

Parameters

node (Node *or* str) – The node to retain after deleting its parent. If a string is passed, the node is determined by finding the node with that label, if any.

Raises

MismatchError – If *node* is a *Node* from a different game.

pygambit.gambit.Game.delete_tree

Game.**delete_tree**(*node:* Node | *str*) → None

Truncate the game tree at *node*, deleting the subtree beneath it.

Parameters

node (Node *or* str) – The node to truncate the game at. If a string is passed, the node is determined by finding the node with that label, if any.

Raises

MismatchError – If *node* is a *Node* from a different game.

Transforming game information structure

Game.set_player(infoset, player)	Set the player at an information set.
Game.set_infoset(node, infoset)	Place *node* in the information set *infoset*.
Game.leave_infoset(node)	Remove this node from its information set.
Game.set_chance_probs(infoset, probs)	Set the action probabilities at chance information set *infoset*.
Game.reveal(infoset, player)	Reveals the move made at *infoset* to *player*.
Game.sort_infosets()	Sort information sets into a standard order.

pygambit.gambit.Game.set_player

Game.**set_player**(*infoset:* Infoset | *str*, *player:* Player | *str*) → None

Set the player at an information set.

Parameters

- **infoset** (Infoset *or* str) – The information set to assign to the player
- **player** (Player *or* str) – The player to have the move at the information set

Raises

MismatchError – If *infoset* is an *Infoset* from another game, or *player* is a *Player* from another game.

pygambit.gambit.Game.set_infoset

Game.**set_infoset**(*node:* Node | *str*, *infoset:* Infoset | *str*) → None

Place *node* in the information set *infoset*. *node* must have the same number of descendants as *infoset* has actions.

Parameters

- **node** (*Node or str*) – The node to set the information set
- **infoset** (*Infoset or str*) – The information set to join

Raises

MismatchError – If *node* is a *Node* from a different game, or *infoset* is an *Infoset* from a different game.

pygambit.gambit.Game.leave_infoset

Game.**leave_infoset**(*node:* Node | *str*)

Remove this node from its information set. If this node is the only node in its information set, this operation has no effect.

Parameters

node (*Node or str*) – The node to move to a new singleton information set.

pygambit.gambit.Game.set_chance_probs

Game.**set_chance_probs**(*infoset:* Infoset | *str*, *probs: Sequence*)

Set the action probabilities at chance information set *infoset*.

Parameters

- **infoset** (*Infoset or str*) – The chance information set at which to set the action probabilities. If a string is passed, the information set is determined by finding the chance information set with that label, if any.
- **probs** (*array-like*) – The action probabilities to set

Raises

- **MismatchError** – If *infoset* is not an information set in this game
- **UndefinedOperationError** – If *infoset* is not an information set of the chance player
- **IndexError** – If the length of *probs* is not the same as the number of actions at the information set
- **ValueError** – If any of the elements of *probs* are not interpretable as numbers, or the values of *probs* are not non-negative numbers that sum to exactly one.

pygambit.gambit.Game.reveal

Game.**reveal**(*infoset:* Infoset | *str*, *player:* Player | *str*) → None

Reveals the move made at *infoset* to *player*.

Revealing the move modifies all subsequent information sets for *player* such that any two nodes which are successors of two different actions at this information set are placed in different information sets for *player*.

Revelation is a one-shot operation; it is not enforced with respect to any revisions made to the game tree subsequently.

Parameters

- **infoset** (*Infoset or str*) – The information set of the move to reveal to the player
- **player** (*Player or str*) – The player to which to reveal the move at this information set.

Raises

MismatchError – If *infoset* is an *Infoset* from a different game, or *player* is a *Player* from a different game.

pygambit.gambit.Game.sort_infosets

Game.**sort_infosets**() → None

Sort information sets into a standard order.

Deprecated since version 16.5.0: This operation is deprecated as efficient management of the iteration orders of information sets and their members is now handled by the representation objects.

Raises
UndefinedOperationError – If the game does not have a tree representation.

Transforming game components

Game.add_player([label])	Add a new player to the game.
Game.add_outcome([payoffs, label])	Add a new outcome to the game.
Game.delete_outcome(outcome)	Delete an outcome from the game.
Game.set_outcome(node, outcome)	Set *outcome* to be the outcome at *node*.
Game.add_strategy(player[, label])	Add a new strategy to the set of strategies for *player*.
Game.delete_strategy(strategy)	Delete *strategy* from the game.

pygambit.gambit.Game.add_player

Game.**add_player**(*label: str = ''*) → *Player*

Add a new player to the game.

Parameters
label (*str, default ""*) – The label for the player.

Returns
A reference to the newly-created player.

Return type
Player

pygambit.gambit.Game.add_outcome

Game.**add_outcome**(*payoffs: list | None = None*, *label: str = ''*) → *Outcome*

Add a new outcome to the game.

Parameters

- **payoffs** (*list, optional*) – The payoffs of the outcome to each player.
- **label** (*str, default ""*) – The label for the outcome

Raises
ValueError – If *payoffs* is specified but is not the same length as the number of players in the game.

Returns
A reference to the newly-created outcome.

Return type
Outcome

pygambit.gambit.Game.delete_outcome

Game.**delete_outcome**(*outcome:* Outcome | *str*) → None

Delete an outcome from the game.

If this game is an extensive game, any node at which this outcome is attached has its outcome reset to null. If this game is a strategic game, any contingency at which this outcome is attached as its outcome reset to null.

Parameters
outcome (*Outcome or str*) – The outcome to delete from the game

Raises
MismatchError – If *outcome* is an *Outcome* from another game.

pygambit.gambit.Game.set_outcome

Game.**set_outcome**(*node:* Node | *str*, *outcome:* Outcome | *str* | *None*) → None

Set *outcome* to be the outcome at *node*. If *outcome* is None, the outcome at *node* is unset.

Parameters

- **node** (*Node or str*) – The node to set the outcome at
- **outcome** (*Outcome or str or None*) – The outcome to assign to the node

Raises
MismatchError – If *node* is a *Node* from a different game, or *outcome* is an *Outcome* from a different game.

pygambit.gambit.Game.add_strategy

Game.**add_strategy**(*player:* Player | *str*, *label: str = None*) → *Strategy*

Add a new strategy to the set of strategies for *player*.

Parameters

- **player** (*Player or str*) – The player to create the new strategy for
- **label** (*str, optional*) – The label to assign to the new strategy

Returns
The newly-created strategy

Return type
Strategy

Raises

- **MismatchError** – If *player* is a *Player* from a different game.
- **UndefinedOperationError** – If called on a game which has an extensive representation.

pygambit.gambit.Game.delete_strategy

Game.**delete_strategy**(*strategy:* Strategy | *str*) → None

Delete *strategy* from the game.

Parameters
strategy (*Strategy or str*) – The strategy to delete

Raises

- **MismatchError** – If *strategy* is a *strategy* from a different game.
- **UndefinedOperationError** – If called on a game which has an extensive representation, or if *strategy* is the only strategy for its player.

Information about the game

Game.title	Get or set the title of the game.
Game.description	Get or set the description of the game.
Game.is_const_sum	Whether the game is constant sum.
Game.is_tree	Return whether a game has a tree-based representation.
Game.is_perfect_recall	Whether the game is perfect recall.
Game.players	The set of players in the game.
Game.outcomes	The set of outcomes in the game.
Game.min_payoff	The minimum payoff to any player in any play of the game.
Game.max_payoff	The maximum payoff to any player in any play of the game.
Game.strategies	The set of strategies in the game.
Game.root	The root node of the game.
Game.actions	The set of actions available in the game.
Game.infosets	The set of information sets in the game.
Game.nodes	The set of nodes in the game.
Game.contingencies	An iterator over the contingencies in the game.

pygambit.gambit.Game.title

Game.**title**

Get or set the title of the game.

The title of the game is an arbitrary string, generally intended to be short.

pygambit.gambit.Game.description

Game.**description**

Get or set the description of the game.

A game's description is an arbitrary string, and may be more discursive than a title.

Changed in version 16.6.0: Renamed `Game.comment` to `Game.description`.

pygambit.gambit.Game.is_const_sum

Game.**is_const_sum**

Whether the game is constant sum.

pygambit.gambit.Game.is_tree

Game.**is_tree**

Return whether a game has a tree-based representation.

pygambit.gambit.Game.is_perfect_recall

Game.**is_perfect_recall**

Whether the game is perfect recall.

By convention, games with a strategic representation have perfect recall as they are treated as simultaneous-move games.

pygambit.gambit.Game.players

Game.**players**

The set of players in the game.

pygambit.gambit.Game.outcomes

Game.**outcomes**

The set of outcomes in the game.

pygambit.gambit.Game.min_payoff

Game.**min_payoff**

The minimum payoff to any player in any play of the game.

Changed in version 16.5.0: Changed from reporting minimum payoff in any (non-null) outcome to the minimum payoff in any play of the game.

See also

Game.max_payoff, *Player.min_payoff*

pygambit.gambit.Game.max_payoff

Game.**max_payoff**

The maximum payoff to any player in any play of the game.

Changed in version 16.5.0: Changed from reporting maximum payoff in any (non-null) outcome to the maximum payoff in any play of the game.

See also

Game.min_payoff, *Player.max_payoff*

pygambit.gambit.Game.strategies

Game.**strategies**

The set of strategies in the game.

pygambit.gambit.Game.root

Game.**root**

The root node of the game.

Raises
UndefinedOperationError – If the game does not hae a tree representation.

pygambit.gambit.Game.actions

Game.**actions**

The set of actions available in the game.

Raises
UndefinedOperationError – If the game does not have a tree representation.

pygambit.gambit.Game.infosets

Game.**infosets**

The set of information sets in the game.

Raises
UndefinedOperationError – If the game does not have a tree representation.

pygambit.gambit.Game.nodes

Game.**nodes**

The set of nodes in the game.

Iteration over this property yields the nodes in the order of depth-first search.

Changed in version 16.4: Changed from a method `nodes()` to a property.

Raises
UndefinedOperationError – If the game does not have a tree representation.

pygambit.gambit.Game.contingencies

Game.**contingencies**

An iterator over the contingencies in the game.

Player.label	Gets or sets the text label of the player.
Player.number	Returns the number of the player in its game.
Player.game	Gets the Game to which the player belongs.
Player.strategies	Returns the set of strategies belonging to the player.
Player.infosets	Returns the set of information sets at which the player has the decision.
Player.actions	Returns the set of actions available to the player at some information set.
Player.is_chance	Returns whether the player is the chance player.
Player.min_payoff	Returns the smallest payoff for the player in any play of the game.
Player.max_payoff	Returns the largest payoff for the player in any play of the game.
Player.strategies	Returns the set of strategies belonging to the player.

pygambit.gambit.Player.label

Player.**label**

Gets or sets the text label of the player.

pygambit.gambit.Player.number

Player.**number**

Returns the number of the player in its game. Players are numbered starting with 0.

pygambit.gambit.Player.game

Player.**game**

Gets the Game to which the player belongs.

pygambit.gambit.Player.strategies

Player.**strategies**

Returns the set of strategies belonging to the player.

pygambit.gambit.Player.infosets

Player.**infosets**

Returns the set of information sets at which the player has the decision.

The iteration order of information sets is the order in which they are encountered in the pre-order depth first traversal of the game tree.

Changed in version 16.5.0: It is no longer necessary to call *Game.sort_infosets* to standardise iteration order.

Raises

UndefinedOperationError – If the game does not have a tree representation.

pygambit.gambit.Player.actions

Player.**actions**

Returns the set of actions available to the player at some information set.

Raises

UndefinedOperationError – If the game does not have a tree representation.

pygambit.gambit.Player.is_chance

Player.**is_chance**

Returns whether the player is the chance player.

pygambit.gambit.Player.min_payoff

Player.**min_payoff**

Returns the smallest payoff for the player in any play of the game.

Changed in version 16.5.0: Changed from reporting minimum payoff in any (non-null) outcome to the minimum payoff in any play of the game.

See also

Player.max_payoff, *Game.min_payoff*

pygambit.gambit.Player.max_payoff

`Player.`**`max_payoff`**

Returns the largest payoff for the player in any play of the game.

Changed in version 16.5.0: Changed from reporting maximum payoff in any (non-null) outcome to the maximum payoff in any play of the game.

See also

Player.min_payoff, *Game.max_payoff*

Outcome.label	The text label associated with this outcome.
Outcome.number	Returns the number of the outcome in the game.
Outcome.game	Returns the game with which this outcome is associated.

pygambit.gambit.Outcome.label

`Outcome.`**`label`**

The text label associated with this outcome.

pygambit.gambit.Outcome.number

`Outcome.`**`number`**

Returns the number of the outcome in the game. Outcomes are numbered starting with 0.

pygambit.gambit.Outcome.game

`Outcome.`**`game`**

Returns the game with which this outcome is associated.

Node.label	The text label associated with the node.
Node.game	Gets the `Game` to which the node belongs.
Node.outcome	Returns the outcome attached to the node.
Node.children	The set of children of this node.
Node.parent	The parent of this node.
Node.is_subgame_root	Returns whether the node is the root of a proper subgame.
Node.is_terminal	Returns whether this is a terminal node of the game.
Node.is_strategy_reachable	Returns whether this node is reachable by any pure strategy profile.
Node.prior_action	The action which leads to this node.
Node.prior_sibling	The node which is immediately before this one in its parent's children.

continues on next page

Table 23 – continued from previous page

Node.next_sibling	The node which is immediately after this one in its parent's children.
Node.infoset	The information set to which this node belongs.
Node.player	The player who makes the decision at this node.
Node.is_successor_of(node)	Returns whether this node is a successor of *node*.
Node.plays	Returns a list of all terminal *Node* objects consistent with it.
Node.own_prior_action	The last action taken by the node's owner before reaching this node.

pygambit.gambit.Node.label

Node.**label**

The text label associated with the node.

pygambit.gambit.Node.game

Node.**game**

Gets the Game to which the node belongs.

pygambit.gambit.Node.outcome

Node.**outcome**

Returns the outcome attached to the node.

If no outcome is attached to the node, None is returned.

pygambit.gambit.Node.children

Node.**children**

The set of children of this node.

pygambit.gambit.Node.parent

Node.**parent**

The parent of this node.

If this is the root node, None is returned.

pygambit.gambit.Node.is_subgame_root

Node.**is_subgame_root**

Returns whether the node is the root of a proper subgame.

Changed in version 16.1.0: Changed to being a property instead of a member function.

pygambit.gambit.Node.is_terminal

Node.**is_terminal**

Returns whether this is a terminal node of the game.

pygambit.gambit.Node.is_strategy_reachable

Node.**is_strategy_reachable**

Returns whether this node is reachable by any pure strategy profile.

A node is considered reachable if there exists at least one pure strategy profile where the resulting path of play passes through that node.

In games with absent-mindedness, some nodes may be unreachable because any path to them requires conflicting choices at the same information set.

pygambit.gambit.Node.prior_action

Node.**prior_action**

The action which leads to this node.

If this is the root node, None is returned.

pygambit.gambit.Node.prior_sibling

Node.**prior_sibling**

The node which is immediately before this one in its parent's children.

If this is the root node or the first child of its parent, None is returned.

pygambit.gambit.Node.next_sibling

Node.**next_sibling**

The node which is immediately after this one in its parent's children.

If this is the root node or the last child of its parent, None is returned.

pygambit.gambit.Node.infoset

Node.**infoset**

The information set to which this node belongs.

If this is a terminal node, which belongs to no information set, None is returned.

pygambit.gambit.Node.player

Node.**player**

The player who makes the decision at this node.

If this is a terminal node, None is returned.

pygambit.gambit.Node.is_successor_of

Node.**is_successor_of**(*node:* Node) → bool

Returns whether this node is a successor of *node*.

pygambit.gambit.Node.plays

Node.**plays**

Returns a list of all terminal *Node* objects consistent with it.

pygambit.gambit.Node.own_prior_action

Node.**own_prior_action**

The last action taken by the node's owner before reaching this node.

Returns

- *Action or None* – The action object, or None if the player has not moved previously on the path to this node.
- *.. versionadded:: 16.5.0*

See also

Infoset.own_prior_actions

Infoset.label	Get or set the text label of the information set.
Infoset.game	The Game to which the information set belongs.
Infoset.is_chance	Whether the information set belongs to the chance player.
Infoset.is_absent_minded	Whether the information set is absent-minded.
Infoset.player	The player who has the move at this information set.
Infoset.actions	The set of actions at the information set.
Infoset.members	The set of nodes which are members of the information set.
Infoset.precedes(node)	Return whether this information set precedes *node* in the game tree.
Infoset.plays	Returns a list of all terminal *Node* objects consistent with it.
Infoset.own_prior_actions	The set of actions taken by the player immediately preceding the member nodes in the information set.

Action.label	Get or set the text label of the action.
Action.infoset	Get the information set to which the action belongs.
Action.precedes(node)	Returns whether *node* precedes this action in the extensive game.
Action.prob	Get the probability a chance action is played.
Action.plays	Returns a list of all terminal *Node* objects consistent with it.

Strategy.label	Get or set the text label associated with the strategy.
Strategy.game	The game to which the strategy belongs.
Strategy.player	The player to which the strategy belongs.
Strategy.number	The number of the strategy.
Strategy.action(infoset)	Get the action prescribed by a strategy for a given information set.

Player behavior

Game.mixed_strategy_profile([data, rational])	Create a mixed strategy profile over the game.
Game.random_strategy_profile([denom, gen])	Create a *MixedStrategy* on the game, with probabilities drawn from the uniform distribution over the set of mixed strategy profiles.
Game.mixed_behavior_profile([data, rational])	Create a mixed behavior profile over the game.
Game.random_behavior_profile([denom, gen])	Create a *MixedBehaviorProfile* on the game, with probabilities drawn from the uniform distribution over the set of mixed behavior profiles.
Game.strategy_support_profile([strategies])	Create a new *StrategySupportProfile* on the game.

pygambit.gambit.Game.mixed_strategy_profile

Game.**mixed_strategy_profile**(*data=None*, *rational=False*) → *MixedStrategyProfile*

Create a mixed strategy profile over the game.

If *data* is not specified, the mixed strategy profile is initialized to uniform randomization for each player over their strategies. If the game has a tree representation, the mixed strategy profile is defined over the reduced strategic form representation.

Parameters

- **data** – A nested list (or compatible type) with the same dimension as the strategy set of the game, specifying the probabilities of the strategies.
- **rational** – If True, probabilities are represented using rational numbers; otherwise floating point numbers are used.

See also

random_strategy_profile
Create a *MixedStrategyProfile* with randomly-drawn probabilities.

pygambit.gambit.Game.random_strategy_profile

Game.**random_strategy_profile**(*denom: int = None*, *gen: Generator | None = None*) → *MixedStrategyProfile*

Create a *MixedStrategy* on the game, with probabilities drawn from the uniform distribution over the set of mixed strategy profiles.

Parameters

- **denom** (*int, optional*) – If specified, the probabilities are generated on a grid with denominator *denom*, and the resulting profile will be a *MixedStrategyProfileRational*. If not specified, the probabilities will be floating point numbers, and the resulting profile will be a *MixedStrategyProfileRational*.
- **gen** (*np.random.Generator, optional*) – If specified, uses the *numpy* random number generator *gen* to generate uniform random samples. Otherwise, uses the default generation method in *numpy*.

- **versionadded:** (..) – 16.2.0: Replaces the functionality of *MixedStrategyProfile.randomize()*.

See also

mixed_strategy_profile
Create a *MixedStrategyProfile* with specified probabilities.

pygambit.gambit.Game.mixed_behavior_profile

Game.**mixed_behavior_profile**(*data=None*, *rational=False*) → *MixedBehaviorProfile*

Create a mixed behavior profile over the game.

If *data* is not specified, the profile is initialized to uniform randomization at each information set.

Parameters

- **data** (*array_like of array_like of array_like, optional*) – A nested list (or compatible type) with the same dimension as the action set of the game, specifying the probabilities of the actions.
- **rational** (*bool, optional*) – If True, probabilities are represented using rational numbers; otherwise floating point numbers are used.

Raises

UndefinedOperationError – If the game does not have a tree representation.

See also

random_behavior_profile
Create a *MixedBehaviorProfile* with randomly-drawn probabilities.

pygambit.gambit.Game.random_behavior_profile

Game.**random_behavior_profile**(*denom: int = None*, *gen: Generator | None = None*) → *MixedBehaviorProfile*

Create a *MixedBehaviorProfile* on the game, with probabilities drawn from the uniform distribution over the set of mixed behavior profiles.

Parameters

- **denom** (*int, optional*) – If specified, the probabilities are generated on a grid with denominator *denom*, and the resulting profile will be a *MixedBehaviorProfileRational*. If not specified, the probabilities will be floating point numbers, and the resulting profile will be a *MixedBehaviorProfileRational*.
- **gen** (*np.random.Generator, optional*) – If specified, uses the *numpy* random number generator *gen* to generate uniform random samples. Otherwise, uses the default generation method in *numpy*.
- **versionadded:** (..) – 16.2.0: Replaces the functionality of *MixedBehaviorProfile.randomize()*.

See also

mixed_behavior_profile
Create a *MixedBehaviorProfile* with specified probabilities.

pygambit.gambit.Game.strategy_support_profile

Game.**strategy_support_profile**(*strategies: Callable | None = None*) → StrategySupportProfile

Create a new *StrategySupportProfile* on the game.

Parameters
strategies (*function, optional*) – By default the support profile contains all strategies for all players. If specified, only strategies for which the supplied function returns *True* are included.

Return type
StrategySupportProfile

Representation of strategic behavior

Probability distributions over strategies

MixedStrategyProfile	Represents a mixed strategy profile over the strategies in a Game.
MixedStrategyProfile.game	The game on which this mixed strategy profile is defined.
MixedStrategyProfile.mixed_strategies()	Iterate over the mixed strategies in the profile.
MixedStrategyProfile.__iter__	Iterate over the probabilities assigned to strategies by the profile.
MixedStrategyProfile.__getitem__	Access a component of the mixed strategy profile specified by *index*.
MixedStrategyProfile.__setitem__	Sets a probability or a mixed strategy to *value*.
MixedStrategyProfile.payoff(player)	Returns the expected payoff to a player if all players play according to the profile.
MixedStrategyProfile.strategy_value(strategy)	Returns the expected payoff to playing the strategy, if all other players play according to the profile.
MixedStrategyProfile.strategy_regret(strategy)	Returns the regret to playing *strategy*, if all other players play according to the profile.
MixedStrategyProfile.player_regret(player)	Returns the regret of *player* for playing their mixed strategy, if all other players play according to the profile.
MixedStrategyProfile.strategy_value_deriv(...)	Returns the derivative of the payoff to playing *strategy*, with respect to the probability that *other* is played.
MixedStrategyProfile.max_regret()	Returns the maximum regret of any player.
MixedStrategyProfile.liap_value()	Returns the Lyapunov value (see [McK91]) of the strategy profile.
MixedStrategyProfile.as_behavior()	Creates a mixed behavior profile which is equivalent to this mixed strategy profile.
MixedStrategyProfile.normalize()	Create a profile with the same strategy proportions as this one, but normalised so probabilities for each player sum to one.
MixedStrategyProfile.copy()	Creates a copy of the mixed strategy profile.
MixedStrategy	A probability distribution over a player's strategies.
MixedStrategy.__iter__	Iterate over the probabilities assigned to strategies by the mixed strategy.

continues on next page

Table 28 – continued from previous page

MixedStrategy.__getitem__	Returns the probability that the strategy referred to by *index* is played.
MixedStrategy.__setitem__	Sets the probability a strategy is played.

pygambit.gambit.MixedStrategyProfile

class pygambit.gambit.**MixedStrategyProfile**

Represents a mixed strategy profile over the strategies in a `Game`.

A mixed strategy profile is a dict-like object, mapping each strategy in a game to the corresponding probability with which that strategy is played.

Mixed strategy profiles may represent probabilities as either exact (rational) numbers, or floating-point numbers. These may not be combined in the same mixed strategy profile.

Changed in version 16.1.0: Profiles are accessed as dict-like objects; indexing by integer player or strategy indices is no longer supported.

See also

Game.mixed_strategy_profile
Creates a new mixed strategy profile on a game.

MixedBehaviorProfile
Represents a mixed behavior profile over a `Game` with an extensive representation.

Methods

as_behavior()	Creates a mixed behavior profile which is equivalent to this mixed strategy profile.
copy()	Creates a copy of the mixed strategy profile.
liap_value()	Returns the Lyapunov value (see [McK91]) of the strategy profile.
max_regret()	Returns the maximum regret of any player.
mixed_strategies()	Iterate over the mixed strategies in the profile.
normalize()	Create a profile with the same strategy proportions as this one, but normalised so probabilities for each player sum to one.
payoff(player)	Returns the expected payoff to a player if all players play according to the profile.
player_regret(player)	Returns the regret of *player* for playing their mixed strategy, if all other players play according to the profile.
strategy_regret(strategy)	Returns the regret to playing *strategy*, if all other players play according to the profile.
strategy_value(strategy)	Returns the expected payoff to playing the strategy, if all other players play according to the profile.
strategy_value_deriv(strategy, other)	Returns the derivative of the payoff to playing *strategy*, with respect to the probability that *other* is played.

Attributes

game	The game on which this mixed strategy profile is defined.

pygambit.gambit.MixedStrategyProfile.game

`MixedStrategyProfile.`**`game`**

The game on which this mixed strategy profile is defined.

pygambit.gambit.MixedStrategyProfile.mixed_strategies

`MixedStrategyProfile.`**`mixed_strategies`**`()` → Iterator[Tuple[*Player*, *MixedStrategy*], None, None]

Iterate over the mixed strategies in the profile.

Added in version 16.2.0.

Yields

- **player** (*Player*) – A player in the game
- **strategy** (*MixedStrategy*) – The player's mixed strategy specified in the profile

pygambit.gambit.MixedStrategyProfile.__iter__

`MixedStrategyProfile.`**`__iter__`**`()`

Iterate over the probabilities assigned to strategies by the profile.

Added in version 16.2.0.

Yields

- **strategy** (*Strategy*) – A strategy in the game
- **probability** (*float or Rational*) – The probability the profile assigns to the strategy being played

pygambit.gambit.MixedStrategyProfile.__getitem__

`MixedStrategyProfile.`**`__getitem__`**`()`

Access a component of the mixed strategy profile specified by *index*.

Parameters

index (`Player, Strategy, or str`) – The part of the profile to return:

- If *index* is a `Player`, returns a `MixedStrategy` over the player's strategies.
- If *index* is a `Strategy`, returns the probability the strategy is played.
- If *index* is a `str`, attempts to resolve the referenced object by first searching for a player with that label, and then for a strategy with that label.

Raises

MismatchError – If *player* is a `Player` from a different game, or *strategy* is a `Strategy` from a different game.

pygambit.gambit.MixedStrategyProfile.__setitem__

MixedStrategyProfile.**__setitem__**()

Sets a probability or a mixed strategy to *value*.

Parameters

- **index** (*Player, Strategy, or str*) – The part of the profile to set:
  - If *index* is a `Player`, sets the `MixedStrategy` over the player's strategies.
  - If *index* is a `Strategy`, sets the probability the strategy is played.
  - If *index* is a `str`, attempts to resolve the referenced object by first searching for a player with that label, and then for a strategy with that label.
- **value** – Any value which can be converted to the data type of the `MixedStrategyProfile`.

Raises

MismatchError – If *player* is a `Player` from a different game, or *strategy* is a `Strategy` from a different game.

pygambit.gambit.MixedStrategyProfile.payoff

MixedStrategyProfile.**payoff**(*player:* Player | *str*) → float | Rational

Returns the expected payoff to a player if all players play according to the profile.

Parameters

player (*Player or str*) – The player to get the payoff for. If a string is passed, the player is determined by finding the player with that label, if any.

Raises

- **MismatchError** – If *player* is a *Player* from a different game.
- **KeyError** – If *player* is a string and no player in the game has that label.

pygambit.gambit.MixedStrategyProfile.strategy_value

MixedStrategyProfile.**strategy_value**(*strategy:* Strategy | *str*) → float | Rational

Returns the expected payoff to playing the strategy, if all other players play according to the profile.

Parameters

strategy (*Strategy or str*) – The strategy to get the payoff for. If a string is passed, the strategy is determined by finding the strategy with that label, if any.

Raises

- **MismatchError** – If *strategy* is a *Strategy* from a different game.
- **KeyError** – If *strategy* is a string and no strategy in the game has that label.

pygambit.gambit.MixedStrategyProfile.strategy_regret

MixedStrategyProfile.**strategy_regret**(*strategy:* Strategy | *str*) → float | Rational

Returns the regret to playing *strategy*, if all other players play according to the profile.

The regret is defined as the difference between the payoff of the best-response strategy and the payoff of *strategy*. By convention, the regret is always non-negative.

Changed in version 16.2.0: Changed from *regret()* to disambiguate from other regret concepts.

Parameters

strategy (*Strategy or str*) – The strategy to get the regret for. If a string is passed, the strategy is determined by finding the strategy with that label, if any.

Raises

- **MismatchError** – If *strategy* is a *Strategy* from a different game.
- **KeyError** – If *strategy* is a string and no strategy in the game has that label.

See also

player_regret, *max_regret*

pygambit.gambit.MixedStrategyProfile.player_regret

MixedStrategyProfile.**player_regret**(*player:* Player | *str*) → float | Rational

Returns the regret of *player* for playing their mixed strategy, if all other players play according to the profile.

The regret is defined as the difference between the payoff of the best-response strategy and the payoff of the player's mixed strategy. By convention, the regret is always non-negative.

Added in version 16.2.0.

Parameters

player (*Player or str*) – The player to get the regret for. If a string is passed, the player is determined by finding the player with that label, if any.

Raises

- **MismatchError** – If *player* is a *Player* from a different game.
- **KeyError** – If *player* is a string and no player in the game has that label.

See also

strategy_regret, *max_regret*

pygambit.gambit.MixedStrategyProfile.strategy_value_deriv

MixedStrategyProfile.**strategy_value_deriv**(*strategy:* Strategy | *str*, *other:* Strategy | *str*) → float | Rational

Returns the derivative of the payoff to playing *strategy*, with respect to the probability that *other* is played.

Raises

- **MismatchError** – If *strategy* or *other* is a *Strategy* from a different game.
- **KeyError** – If *strategy* or *other* is a string and no strategy in the game has that label.

pygambit.gambit.MixedStrategyProfile.max_regret

MixedStrategyProfile.**max_regret**() → float | Rational

Returns the maximum regret of any player.

A profile is a Nash equilibrium if and only if *max_regret()* is 0.

Added in version 16.2.0.

> **See also**
>
> *strategy_regret*, *player_regret*, *liap_value*

pygambit.gambit.MixedStrategyProfile.liap_value

MixedStrategyProfile.**liap_value**() → float | Rational

Returns the Lyapunov value (see [McK91]) of the strategy profile.

The Lyapunov value is a non-negative number which is zero exactly at Nash equilibria.

> **See also**
>
> *max_regret*

pygambit.gambit.MixedStrategyProfile.as_behavior

MixedStrategyProfile.**as_behavior**() → *MixedBehaviorProfile*

Creates a mixed behavior profile which is equivalent to this mixed strategy profile.

Returns
The equivalent mixed behavior profile.

Return type
MixedBehaviorProfile

Raises
UndefinedOperationError – If the game does not have a tree representation.

pygambit.gambit.MixedStrategyProfile.normalize

MixedStrategyProfile.**normalize**() → *MixedStrategyProfile*

Create a profile with the same strategy proportions as this one, but normalised so probabilities for each player sum to one. Requires that all players have non-negative entries that are not all equal to zero.

Returns
The normalized mixed strategy profile.

Return type
MixedStrategyProfile

Raises
ValueError – If the input mixed strategy of any player is all zero or has a negative entry.

pygambit.gambit.MixedStrategyProfile.copy

MixedStrategyProfile.**copy**() → *MixedStrategyProfile*

Creates a copy of the mixed strategy profile.

pygambit.gambit.MixedStrategy

class pygambit.gambit.**MixedStrategy**

A probability distribution over a player's strategies.

A `MixedStrategy` represents the component of a `MixedStrategyProfile` associated with a given `Player`. The full profile is accessible via the *profile* attribute, and the player for whom the `MixedStrategy` applies is accessible via *player*.

Methods

Attributes

`player`	The player for whom this mixed strategy is defined.
`profile`	The full profile of which this is a part.

pygambit.gambit.MixedStrategy.__iter__

`MixedStrategy.__iter__()`

Iterate over the probabilities assigned to strategies by the mixed strategy.

Added in version 16.2.0.

Yields

- **strategy** (*Strategy*) – A strategy for the player
- **probability** (*float or Rational*) – The probability the mixed strategy assigns to the strategy being played

pygambit.gambit.MixedStrategy.__getitem__

`MixedStrategy.__getitem__()`

Returns the probability that the strategy referred to by *index* is played.

Parameters

index (`Strategy` *or* `str`) –

- If *index* is a `Strategy`, returns the probability the strategy is played.
- If *index* is a `str`, attempts to resolve the referenced object by searching for a strategy with that label.

Returns

The probability assigned to the strategy.

Return type

float or Rational

Raises

MismatchError – If *index* is a `Strategy` that does not belong to this `MixedStrategy`'s player.

pygambit.gambit.MixedStrategy.__setitem__

`MixedStrategy.__setitem__()`

Sets the probability a strategy is played.

Parameters

- **index** (`Strategy`, *or* `str`) – The part of the profile to set:

- If *index* is a `Strategy`, sets the probability the strategy is played.
- If *index* is a `str`, attempts to resolve the referenced object by searching for a strategy with that label, and sets the probability for that strategy.

- **value** – Any value which can be converted to the data type of the `MixedStrategyProfile`.

Raises

MismatchError – If *strategy* is a `Strategy` that does not belong to this `MixedStrategy`'s player.

Probability distributions over behavior

MixedBehaviorProfile	Represents a mixed behavior profile over the actions in a `Game`.
MixedBehaviorProfile.game	The game on which this mixed behavior profile is defined.
MixedBehaviorProfile.mixed_behaviors()	Iterate over the mixed behaviors in the profile.
MixedBehaviorProfile.mixed_actions()	Iterate over the mixed actions specified by the profile.
MixedBehaviorProfile.__iter__	Iterate over the probabilities assigned to actions by the profile.
MixedBehaviorProfile.__getitem__	Access a component of the mixed behavior specified by *index*.
MixedBehaviorProfile.__setitem__	Sets a probability, mixed agent strategy, or mixed behavior strategy to *value*.
MixedBehaviorProfile.payoff(player)	Returns the expected payoff to a player if all players play according to the profile.
MixedBehaviorProfile.action_value(action)	Returns the expected payoff to the player of playing an action conditional on reaching its information set, if all players play according to the profile.
MixedBehaviorProfile.action_regret(action)	Returns the regret to playing *action*, if all other players play according to the profile.
MixedBehaviorProfile.infoset_value(infoset)	Returns the expected payoff to the player conditional on reaching an information set, if all players play according to the profile.
MixedBehaviorProfile.infoset_regret(infoset)	Returns the regret to the player for playing their mixed action at *infoset*, if all other players play according to the profile.
MixedBehaviorProfile.node_value(player, node)	Returns the expected payoff to *player* conditional on play reaching *node*, if all players play according to the profile.
MixedBehaviorProfile.realiz_prob(node)	Returns the probability with which a node is reached.
MixedBehaviorProfile.infoset_prob(infoset)	Returns the probability with which an information set is reached.
MixedBehaviorProfile.belief(node)	Returns the conditional probability that a node is reached, given that its information set is reached.
MixedBehaviorProfile.is_defined_at(infoset)	Returns whether the profile has probabilities defined at the information set.
MixedBehaviorProfile.agent_max_regret()	Returns the maximum regret at any information set.
MixedBehaviorProfile.agent_liap_value()	Returns the Lyapunov value (see [McK91]) of the strategy profile.
MixedBehaviorProfile.max_regret()	Returns the maximum regret at any information set.
MixedBehaviorProfile.liap_value()	Returns the Lyapunov value (see [McK91]) of the strategy profile.

continues on next page

Table 33 – continued from previous page

MixedBehaviorProfile.as_strategy()	Returns a *MixedStrategyProfile* which is equivalent to the profile.
MixedBehaviorProfile.normalize()	Create a profile with the same action proportions as this one, but normalised so probabilities for each infoset sum to one.
MixedBehaviorProfile.copy()	Creates a copy of the behavior strategy profile.
MixedBehavior	A set of probability distributions describing a player's behavior.
MixedBehavior.mixed_actions()	Iterate over the mixed actions specified by the mixed behavior.
MixedBehavior.__iter__	Iterate over the probabilities assigned to actions by the mixed behavior.
MixedBehavior.__getitem__	Access a component of the mixed behavior specified by *index*.
MixedBehavior.__setitem__	Sets a component of the mixed behavior to *value*.
MixedAction	A probability distribution over a player's actions at an information set.
MixedAction.__iter__	Iterate over the probabilities assigned to actions by the mixed action.
MixedAction.__getitem__	Returns the probability that the action referred to by *index* is played.
MixedAction.__setitem__	Sets the probability an action is played.

pygambit.gambit.MixedBehaviorProfile

class pygambit.gambit.**MixedBehaviorProfile**

Represents a mixed behavior profile over the actions in a `Game`.

A mixed behavior profile is a dict-like object, mapping each action at each information set in a game to the corresponding probability with which the action is played, conditional on that information set being reached.

Mixed behavior profiles may represent probabilities as either exact (rational) numbers, or floating-point numbers. These may not be combined in the same mixed behavior profile.

Changed in version 16.1.0: Profiles are accessed as dict-like objects; indexing by integer player, infoset, or action indices is no longer supported.

See also

Game.mixed_behavior_profile
Creates a new mixed behavior profile on a game.

MixedStrategyProfile
Represents a mixed strategy profile over a `Game`.

Methods

action_regret(action)	Returns the regret to playing *action*, if all other players play according to the profile.

continues on next page

Table 34 – continued from previous page

action_value(action)	Returns the expected payoff to the player of playing an action conditional on reaching its information set, if all players play according to the profile.
agent_liap_value()	Returns the Lyapunov value (see [McK91]) of the strategy profile.
agent_max_regret()	Returns the maximum regret at any information set.
as_strategy()	Returns a *MixedStrategyProfile* which is equivalent to the profile.
belief(node)	Returns the conditional probability that a node is reached, given that its information set is reached.
copy()	Creates a copy of the behavior strategy profile.
infoset_prob(infoset)	Returns the probability with which an information set is reached.
infoset_regret(infoset)	Returns the regret to the player for playing their mixed action at *infoset*, if all other players play according to the profile.
infoset_value(infoset)	Returns the expected payoff to the player conditional on reaching an information set, if all players play according to the profile.
is_defined_at(infoset)	Returns whether the profile has probabilities defined at the information set.
liap_value()	Returns the Lyapunov value (see [McK91]) of the strategy profile.
max_regret()	Returns the maximum regret at any information set.
mixed_actions()	Iterate over the mixed actions specified by the profile.
mixed_behaviors()	Iterate over the mixed behaviors in the profile.
node_value(player, node)	Returns the expected payoff to *player* conditional on play reaching *node*, if all players play according to the profile.
normalize()	Create a profile with the same action proportions as this one, but normalised so probabilities for each infoset sum to one.
payoff(player)	Returns the expected payoff to a player if all players play according to the profile.
realiz_prob(node)	Returns the probability with which a node is reached.

Attributes

game	The game on which this mixed behavior profile is defined.

pygambit.gambit.MixedBehaviorProfile.game

MixedBehaviorProfile.**game**

The game on which this mixed behavior profile is defined.

pygambit.gambit.MixedBehaviorProfile.mixed_behaviors

MixedBehaviorProfile.**mixed_behaviors**() → Iterator[tuple[*Player*, *MixedBehavior*], None, None]

Iterate over the mixed behaviors in the profile.

Added in version 16.2.0.

Yields

- **player** (*Player*) – A player in the game
- **behavior** (*MixedBehavior*) – The player's mixed behavior specified in the profile

pygambit.gambit.MixedBehaviorProfile.mixed_actions

MixedBehaviorProfile.**mixed_actions**() → Iterator[tuple[*Infoset*, *MixedAction*], None, None]

Iterate over the mixed actions specified by the profile.

Added in version 16.2.0.

Yields

- **infoset** (*Infoset*) – An information set in the game
- **action** (*MixedAction*) – The mixed action specified at the information set by the profile.

pygambit.gambit.MixedBehaviorProfile.__iter__

MixedBehaviorProfile.**__iter__**()

Iterate over the probabilities assigned to actions by the profile.

Added in version 16.2.0.

Yields

- **action** (*Action*) – An action in the game
- **probability** (*float or Rational*) – The probability the profile assigns to the action being played

pygambit.gambit.MixedBehaviorProfile.__getitem__

MixedBehaviorProfile.**__getitem__**()

Access a component of the mixed behavior specified by *index*.

Parameters

index (`Player, Infoset, Action, or str`) – The part of the profile to return:

- If *index* is a `Player`, returns a `MixedBehavior` over the player's infosets
- If *index* is an `Infoset`, returns a `MixedAction` over the infoset's actions
- If *index* is an `Action`, returns the probability the action is played
- If *index* is a `str`, attempts to resolve the referenced object by first searching for a player with that label, then for an infoset with that label, and finally for an action with that label.

Raises

MismatchError – If *player* is a `Player` from a different game, *infoset* is an `Infoset` from a different game, or *action* is an `Action` from a different game.`

pygambit.gambit.MixedBehaviorProfile.__setitem__

MixedBehaviorProfile.__setitem__()

Sets a probability, mixed agent strategy, or mixed behavior strategy to *value*.

Parameters

index (Player, Infoset, Action, *or str*) – The part of the profile to return:

- If *index* is a Player, sets the MixedBehavior over the player's infosets
- If *index* is an Infoset, sets the MixedAction over the infoset's actions
- If *index* is an Action, sets the probability the action is played
- If *index* is a str, attempts to resolve the referenced object by first searching for a player with that label, then for an infoset with that label, and finally for an action with that label.

Raises

MismatchError – If *player* is a Player from a different game, *infoset* is an Infoset from a different game, or *action* is an Action from a different game.`

pygambit.gambit.MixedBehaviorProfile.payoff

MixedBehaviorProfile.**payoff**(*player:* Player | *str*) → float | Rational

Returns the expected payoff to a player if all players play according to the profile.

Parameters

player (Player *or str*) – The player to get the payoff for. If a string is passed, the player is determined by finding the player with that label, if any.

Raises

- **MismatchError** – If *player* is a Player from a different game.
- **KeyError** – If *player* is a string and no player in the game has that label.
- **ValueError** – If *player* resolves to the chance player

pygambit.gambit.MixedBehaviorProfile.action_value

MixedBehaviorProfile.**action_value**(*action:* Action | *str*) → float | Rational | None

Returns the expected payoff to the player of playing an action conditional on reaching its information set, if all players play according to the profile.

If the information set is not reachable, the expected payoff is not well-defined. In this case, the function returns *None*.

Parameters

action (Action *or str*) – The action to get the payoff for. If a string is passed, the action is determined by finding the action with that label, if any.

Raises

- **MismatchError** – If *action* is an Action from a different game.
- **KeyError** – If *action* is a string and no action in the game has that label.
- **ValueError** – If *action* resolves to an action that belongs to the chance player

See also

MixedBehaviorProfile.infoset_prob

pygambit.gambit.MixedBehaviorProfile.action_regret

MixedBehaviorProfile.**action_regret**(*action:* Action | *str*) → float | Rational

Returns the regret to playing *action*, if all other players play according to the profile.

The regret is defined as the difference between the payoff of the best-response action and the payoff of *action*. Payoffs are computed conditional on reaching the information set. By convention, the regret is always non-negative.

Changed in version 16.2.0: Changed from *regret()* to disambiguate from other regret concepts.

Parameters

action (*Action or str*) – The action to get the regret for. If a string is passed, the action is determined by finding the action with that label, if any.

Raises

- **MismatchError** – If *action* is an `Action` from a different game.
- **KeyError** – If *action* is a string and no action in the game has that label.

See also

infoset_regret, *max_regret*

pygambit.gambit.MixedBehaviorProfile.infoset_value

MixedBehaviorProfile.**infoset_value**(*infoset:* Infoset | *str*) → float | Rational | None

Returns the expected payoff to the player conditional on reaching an information set, if all players play according to the profile.

If the information set is not reachable, the expected payoff is not well-defined. In this case, the function returns *None*.

Parameters

infoset (*Infoset or str*) – The information set to get the payoff for. If a string is passed, the information set is determined by finding the information set with that label, if any.

Raises

- **MismatchError** – If *infoset* is an `Infoset` from a different game.
- **KeyError** – If *infoset* is a string and no information set in the game has that label.
- **ValueError** – If *infoset* resolves to an infoset that belongs to the chance player

See also

MixedBehaviorProfile.infoset_prob

pygambit.gambit.MixedBehaviorProfile.infoset_regret

MixedBehaviorProfile.**infoset_regret**(*infoset:* Infoset | *str*) → float | Rational

Returns the regret to the player for playing their mixed action at *infoset*, if all other players play according to the profile.

The regret is defined as the difference between the payoff of the best-response action and the payoff of the player's mixed action. Payoffs are computed conditional on reaching the information set. By convention, the regret is always non-negative.

Added in version 16.2.0.

Parameters

infoset (*Infoset or str*) – The information set to get the regret at. If a string is passed, the information set is determined by finding the information set with that label, if any.

Raises

- **MismatchError** – If *infoset* is an Infoset from a different game.
- **KeyError** – If *infoset* is a string and no information set in the game has that label.

See also

action_regret, *agent_max_regret*

pygambit.gambit.MixedBehaviorProfile.node_value

MixedBehaviorProfile.**node_value**(*player:* Player | *str*, *node:* Node | *str*) → float | Rational

Returns the expected payoff to *player* conditional on play reaching *node*, if all players play according to the profile.

Parameters

- **player** (*Player or str*) – The player to get the payoff for. If a string is passed, the player is determined by finding the player with that label, if any.
- **node** (*Node or str*) – The node to get the payoff at. If a string is passed, the node is determined by finding the node with that label, if any.

Raises

- **MismatchError** – If *player* is a Player from a different game or *node* is a Node from a different game.
- **KeyError** – If *player* is a string and no player in the game has that label, or *node* is a string and no node in the game has that label.
- **ValueError** – If *player* resolves to the chance player

pygambit.gambit.MixedBehaviorProfile.realiz_prob

MixedBehaviorProfile.**realiz_prob**(*node:* Node | *str*) → float | Rational

Returns the probability with which a node is reached.

Parameters

node (*Node or str*) – The node to get the payoff for. If a string is passed, the node is determined by finding the node with that label, if any.

Raises

- **MismatchError** – If *node* is a Node from a different game.
- **KeyError** – If *node* is a string and no node in the game has that label.

pygambit.gambit.MixedBehaviorProfile.infoset_prob

MixedBehaviorProfile.**infoset_prob**(*infoset:* Node | *str*) → float | Rational

Returns the probability with which an information set is reached.

Parameters

infoset (Infoset *or* str) – The information set to get the payoff for. If a string is passed, the information set is determined by finding the information set with that label, if any.

Raises

- **MismatchError** – If *infoset* is an Infoset from a different game.
- **KeyError** – If *infoset* is a string and no information set in the game has that label.

pygambit.gambit.MixedBehaviorProfile.belief

MixedBehaviorProfile.**belief**(*node:* Node | *str*) → float | Rational | None

Returns the conditional probability that a node is reached, given that its information set is reached.

If the information set is not reachable, the belief is not well-defined. In this case, the function returns *None*.

Parameters

node – The node of the game tree

Raises

MismatchError – If *node* is not in the same game as the profile

See also

MixedBehaviorProfile.infoset_prob

pygambit.gambit.MixedBehaviorProfile.is_defined_at

MixedBehaviorProfile.**is_defined_at**(*infoset:* Infoset | *str*) → bool

Returns whether the profile has probabilities defined at the information set. A profile can be well-defined if probabilities are not specified at some information sets, as long as those information sets are reached with zero probability.

Parameters

infoset (Infoset *or* str) – The information set to check. If a string is passed, the information set is determined by finding the information set with that label, if any.

Raises

- **MismatchError** – If *infoset* is an Infoset from a different game.
- **KeyError** – If *infoset* is a string and no information set in the game has that label.

pygambit.gambit.MixedBehaviorProfile.agent_max_regret

MixedBehaviorProfile.**agent_max_regret**() → float | Rational

Returns the maximum regret at any information set.

A profile is an agent Nash equilibrium if and only if *agent_max_regret()* is 0.

Changed in version 16.5.0: Renamed from *max_regret* to *agent_max_regret* to clarify the distinction between per-player and per-agent concepts.

See also

action_regret, *infoset_regret*, *max_regret*, *agent_liap_value*

pygambit.gambit.MixedBehaviorProfile.agent_liap_value

MixedBehaviorProfile.**agent_liap_value**() → float | Rational

Returns the Lyapunov value (see [McK91]) of the strategy profile.

The agent Lyapunov value is a non-negative number which is zero exactly at agent Nash equilibria.

Changed in version 16.5.0: Renamed from *liap_value* to *agent_liap_value* to clarify the distinction between per-player and per-agent concepts.

See also

agent_max_regret, *liap_value*

pygambit.gambit.MixedBehaviorProfile.max_regret

MixedBehaviorProfile.**max_regret**() → float | Rational

Returns the maximum regret at any information set.

A profile is a Nash equilibrium if and only if *max_regret()* is 0.

Changed in version 16.5.0: New implementation of *max_regret* to clarify the distinction between per-player and per-agent concepts.

See also

liap_value, *agent_max_regret*

pygambit.gambit.MixedBehaviorProfile.liap_value

MixedBehaviorProfile.**liap_value**() → float | Rational

Returns the Lyapunov value (see [McK91]) of the strategy profile.

The Lyapunov value is a non-negative number which is zero exactly at Nash equilibria.

Changed in version 16.5.0: New implementation of *liap_value* to clarify the distinction between per-player and per-agent concepts.

See also

max_regret, *agent_liap_value*

pygambit.gambit.MixedBehaviorProfile.as_strategy

MixedBehaviorProfile.**as_strategy**() → *MixedStrategyProfile*

Returns a *MixedStrategyProfile* which is equivalent to the profile.

pygambit.gambit.MixedBehaviorProfile.normalize

MixedBehaviorProfile.**normalize**() → *MixedBehaviorProfile*

Create a profile with the same action proportions as this one, but normalised so probabilities for each infoset sum to one.

pygambit.gambit.MixedBehaviorProfile.copy

MixedBehaviorProfile.**copy**() → *MixedBehaviorProfile*

Creates a copy of the behavior strategy profile.

pygambit.gambit.MixedBehavior

class pygambit.gambit.**MixedBehavior**

A set of probability distributions describing a player's behavior.

A `MixedBehavior` represents the component of a `MixedBehaviorProfile` associated with a given `Player`. The full profile is accessible via the *profile* attribute, and the player for whom the `MixedBehavior` applies is accessible via *player*.

Methods

mixed_actions()	Iterate over the mixed actions specified by the mixed behavior.

Attributes

player	The player for whom this mixed behavior strategy is defined.
profile	The full profile of which this is a part.

pygambit.gambit.MixedBehavior.mixed_actions

MixedBehavior.**mixed_actions**() → Iterator[tuple[*Infoset*, *MixedAction*], None, None]

Iterate over the mixed actions specified by the mixed behavior.

Added in version 16.2.0.

Yields

- **infoset** (*Infoset*) – An information set belonging to the player
- **action** (*MixedAction*) – The player's mixed action specified in the mixed behavior

pygambit.gambit.MixedBehavior.__iter__

MixedBehavior.__iter__()

Iterate over the probabilities assigned to actions by the mixed behavior.

Added in version 16.2.0.

Yields

- **action** (*Action*) – An action for the player
- **probability** (*float or Rational*) – The probability the behavior assigns to the action being played

pygambit.gambit.MixedBehavior.__getitem__

MixedBehavior.__getitem__()

Access a component of the mixed behavior specified by *index*.

Parameters

index (Infoset, Action, or str) – The part of the mixed behavior to return:

- If *index* is an Infoset, returns a MixedAction over the infoset's actions
- If *index* is an Action, returns the probability the action is played
- If *index* is a str, attempts to resolve the referenced object by first searching for an infoset with that label, and then for an action with that label.

Raises

MismatchError – If *infoset* not an Infoset for the mixed behavior's player, or *action* is not an Action for the mixed behavior's player.

pygambit.gambit.MixedBehavior.__setitem__

MixedBehavior.__setitem__()

Sets a component of the mixed behavior to *value*.

Parameters

index (Infoset, Action, or str) – The component of the mixed behavior to set:

- If *index* is an *Infoset*, sets the mixed action over that infoset's actions
- If *index* is an *Action*, sets the probability the action is played
- If *index* is a *str*, attempts to resolve the referenced object by first searching for an infoset with that label, and then for an action with that label.

Raises

MismatchError – If *infoset* not an Infoset for the mixed behavior's player, or *action* is not an Action for the mixed behavior's player.

pygambit.gambit.MixedAction

class pygambit.gambit.MixedAction

A probability distribution over a player's actions at an information set.

A MixedAction represents a component of a MixedBehaviorProfile. The full profile is accessible via the *profile* attribute, and the information set at which the MixedAction applies is accessible via *infoset*.

Methods

Attributes

infoset	The information set over which this mixed action is defined.
profile	The full profile of which this is a part.

pygambit.gambit.MixedAction.__iter__

MixedAction.__iter__()

Iterate over the probabilities assigned to actions by the mixed action.

Added in version 16.2.0.

Yields

- **action** (*Action*) – An action at the information set
- **probability** (*float or Rational*) – The probability the mixed action assigns to the action being played

pygambit.gambit.MixedAction.__getitem__

MixedAction.__getitem__()

Returns the probability that the action referred to by *index* is played.

Parameters

index (*Action or str*) –

- If *index* is an Action, returns the probability the action is played.
- If *index* is a str, attempts to resolve the referenced object by searching for an action with that label.

Returns

The probability assigned to the action.

Return type

float or Rational

Raises

MismatchError – If *index* is an Action that does not belong to this MixedAction's information set.

pygambit.gambit.MixedAction.__setitem__

MixedAction.__setitem__()

Sets the probability an action is played.

Parameters

- **index** (*Action or str*) – The part of the profile to set:
  - If *index* is an Action, sets the probability the action is played.

- If *index* is a `str`, attempts to resolve the referenced object by searching for an action with that label, and sets the probability for that action.

- **value** – Any value which can be converted to the data type of the `MixedBehaviorProfile`.

Raises

MismatchError – If *action* is an `Action` that does not belong to this `MixedAction`'s information set.

Computation on supports

undominated_strategies_solve(profile[, ...])	Return a support profile including only the strategies in *profile* which are not dominated by another pure strategy.

pygambit.supports.undominated_strategies_solve

pygambit.supports.**undominated_strategies_solve**(*profile:* Game | *StrategySupportProfile*, *strict: bool = False*, *external: bool = False*) → StrategySupportProfile

Return a support profile including only the strategies in *profile* which are not dominated by another pure strategy.

This function performs only one round of elimination.

Parameters

- **profile** (*Game or StrategySupportProfile*) – The initial profile of strategies. If a *Game* is passed, elimination begins with the full set of strategies on the game.
- **strict** (*bool, default False*) – If specified *True*, eliminate only strategies which are strictly dominated. If *False*, strategies which are weakly dominated are also eliminated.
- **external** (*bool, default False*) – The default is to consider dominance only by strategies which are in the support profile for that player. If *True*, strategies which are dominated by another strategy not in the support profile are also eliminated.

Returns

A new support profile containing only the strategies which are not dominated.

Return type

StrategySupportProfile

Computation of Nash equilibria

NashComputationResult(game, method, ...)	Represents the result of a method which computes Nash equilibria in a game.
enumpure_solve(game)	Compute all *pure-strategy Nash equilibria* of game.
enumpure_agent_solve(game)	Compute all *pure-strategy agent Nash equilibria* of game.
enummixed_solve(game[, rational, lrsnash_path])	Compute all *mixed-strategy Nash equilibria* of a two-player game using the strategic representation.
enumpoly_solve(game[, use_strategic, ...])	Compute Nash equilibria by enumerating all support profiles of strategies or actions, and for each support finding all totally-mixed equilibria of the game over that support.

continues on next page

Table 41 – continued from previous page

lp_solve(game[, rational, use_strategic])	Compute Nash equilibria of a two-player constant-sum game using *linear programming*.
lcp_solve(game[, rational, use_strategic, ...])	Compute Nash equilibria of a two-player game using *linear complementarity programming*.
liap_solve(start[, maxregret, maxiter])	Compute approximate Nash equilibria of a game using *Lyapunov function minimization*.
liap_agent_solve(start[, maxregret, maxiter])	Compute approximate agent Nash equilibria of a game using *Lyapunov function minimization*.
logit_solve(game[, use_strategic, ...])	Compute Nash equilibria of a game using *the logit quantal response equilibrium correspondence*.
simpdiv_solve(start[, maxregret, refine, leash])	Compute Nash equilibria of a game using *simplicial subdivision*.
ipa_solve(perturbation)	Compute Nash equilibria of a game using *iterated polymatrix approximation*.
gnm_solve(perturbation[, end_lambda, steps, ...])	Compute Nash equilibria of a game using *a global Newton method*.

pygambit.nash.NashComputationResult

class pygambit.nash.**NashComputationResult**(*game: ~pygambit.gambit.Game, method: str, rational: bool, use_strategic: bool, equilibria: list[~pygambit.gambit.MixedStrategyProfile] | list[~pygambit.gambit.MixedBehaviorProfile], parameters: dict = <factory>*)

Represents the result of a method which computes Nash equilibria in a game.

game

The game on which the method was run.

Type
Game

method

A string indicating the name of the method used.

Type
str

rational

Whether the calculation used exact rational arithmetic (True) or floating-point (False).

Type
bool

use_strategic

Whether the method solved using the strategic representation (True) or the extensive representation (False).

Type
bool

equilibria

The list of equilibrium profiles computed.

Type
MixedStrategyEquilibriumSet or MixedBehaviorEquilibriumSet

parameters
A dictionary recording any additional algorithm parameters used.

Type
dict

Methods

Attributes

game
method
rational
use_strategic
equilibria
parameters

pygambit.nash.enumpure_solve

pygambit.nash.**enumpure_solve**(*game:* Game) → *NashComputationResult*
Compute all *pure-strategy Nash equilibria* of game.

Changed in version 16.5.0:

***use_strategic* parameter removed. The old behavior in the case**
of *use_strategic=False* is now available as *enumpure_agent_solve*.

Parameters
game (Game) – The game to compute equilibria in.

Returns
res – The result represented as a NashComputationResult object.

Return type
NashComputationResult

See also

enumpure_agent_solve

pygambit.nash.enumpure_agent_solve

pygambit.nash.**enumpure_agent_solve**(*game:* Game) → *NashComputationResult*
Compute all *pure-strategy agent Nash equilibria* of game.

Parameters
game (Game) – The game to compute agent-Nash equilibria in.

Returns
res – The result represented as a NashComputationResult object.

Return type
NashComputationResult

See also

enumpure_solve

pygambit.nash.enummixed_solve

pygambit.nash.**enummixed_solve**(*game:* Game, *rational: bool = True*, *lrsnash_path: Path | str | None = None*) → *NashComputationResult*

Compute all *mixed-strategy Nash equilibria* of a two-player game using the strategic representation.

Parameters

- **game** (Game) – The game to compute equilibria in.
- **rational** (*bool, default True*) – Compute using rational numbers. If *False*, using floating-point arithmetic. Using rationals is more precise, but slower.
- **lrsnash_path** (*pathlib.Path | str | None = None,*) – If specified, use lrsnash to solve the systems of equations. This argument specifies the path to the lrsnash executable.

  Added in version 16.3.0.

Returns

res – The result represented as a `NashComputationResult` object.

Return type

NashComputationResult

Raises

RuntimeError – If game has more than two players.

Notes

lrsnash is part of *lrslib*, available at http://cgm.cs.mcgill.ca/~avis/C/lrs.html

pygambit.nash.enumpoly_solve

pygambit.nash.**enumpoly_solve**(*game:* Game, *use_strategic: bool = False*, *stop_after: int | None = None*, *maxregret: float = 0.0001*, *phcpack_path: Path | str | None = None*) → *NashComputationResult*

Compute Nash equilibria by enumerating all support profiles of strategies or actions, and for each support finding all totally-mixed equilibria of the game over that support.

Parameters

- **game** (Game) – The game to compute equilibria in.
- **use_strategic** (*bool, default False*) – Whether to use the strategic form. If *True*, always uses the strategic representation even if the game's native representation is extensive.
- **stop_after** (*int, optional*) – Maximum number of equilibria to compute. If not specified, examines all support profiles of the game.
- **maxregret** (*float, default 1e-4*) – The acceptance criterion for approximate Nash equilibrium; the maximum regret of any player must be no more than *maxregret* times the difference of the maximum and minimum payoffs of the game
- **phcpack_path** (*str or pathlib.Path, optional*) – If specified, use PHCpack to solve the systems of equations. This argument specifies the path to the PHCpack executable. With this method, only enumeration on the strategic game is supported.

Returns
res – The result represented as a `NashComputationResult` object.

Return type
NashComputationResult

Notes

PHCpack is available at https://homepages.math.uic.edu/~jan/PHCpack/phcpack.html

pygambit.nash.lp_solve

pygambit.nash.**lp_solve**(*game: Game*, *rational: bool = True*, *use_strategic: bool = False*) → *NashComputationResult*

Compute Nash equilibria of a two-player constant-sum game using *linear programming*.

Parameters

- **game** (*Game*) – The game to compute equilibria in.
- **rational** (*bool, default True*) – Compute using rational numbers. If *False*, using floating-point arithmetic. Using rationals is more precise, but slower.
- **use_strategic** (*bool, default False*) – Whether to use the strategic form. If *True*, always uses the strategic representation even if the game's native representation is extensive.

Returns
res – The result represented as a `NashComputationResult` object.

Return type
NashComputationResult

Raises
RuntimeError – If game has more than two players or is not constant sum.

pygambit.nash.lcp_solve

pygambit.nash.**lcp_solve**(*game: Game*, *rational: bool = True*, *use_strategic: bool = False*, *stop_after: int | None = None*, *max_depth: int | None = None*) → *NashComputationResult*

Compute Nash equilibria of a two-player game using *linear complementarity programming*.

Parameters

- **game** (*Game*) – The game to compute equilibria in.
- **rational** (*bool, default True*) – Compute using rational numbers. If *False*, using floating-point arithmetic. Using rationals is more precise, but slower.
- **use_strategic** (*bool, default False*) – Whether to use the strategic form. If *True*, always uses the strategic representation even if the game's native representation is extensive.
- **stop_after** (*int, optional*) – Maximum number of equilibria to compute when using the strategic representation. If not specified, computes all accessible equilibria.
- **max_depth** (*int, optional*) – Maximum depth of recursion when using the strategic representation. If specified, will limit the recursive search, but may result in some accessible equilibria not being found.

Returns
res – The result represented as a `NashComputationResult` object.

Return type
NashComputationResult

Raises

- **RuntimeError** – If game has more than two players.
- **ValueError** – If stop_after or max_depth are supplied for use on the tree representation.

pygambit.nash.liap_solve

pygambit.nash.**liap_solve**(*start: MixedStrategyProfileDouble*, *maxregret: float = 0.0001*, *maxiter: int = 1000*) → *NashComputationResult*

Compute approximate Nash equilibria of a game using *Lyapunov function minimization*.

Changed in version 16.2.0: Method now takes a starting point (as a mixed strategy or mixed behavior profile) instead of a game. Implemented *maxregret* to specify acceptance criterion for approximation.

Changed in version 16.5.0: Computing agent Nash equilibria in the extensive game moved to *liap_agent_solve* for clarity.

Parameters

- **start** (*MixedStrategyProfileDouble*) – The starting profile for function minimization. Up to one equilibrium will be found from any starting profile, and the equilibrium found may (and generally will) depend on the initial profile chosen.
- **maxregret** (*float, default 1e-4*) – The acceptance criterion for approximate Nash equilibrium; the maximum regret of any player must be no more than *maxregret* times the difference of the maximum and minimum payoffs of the game
- **maxiter** (*int, default 1000*) – Maximum number of iterations in function minimization.

Returns
res – The result represented as a NashComputationResult object.

Return type
NashComputationResult

pygambit.nash.liap_agent_solve

pygambit.nash.**liap_agent_solve**(*start: MixedBehaviorProfileDouble*, *maxregret: float = 0.0001*, *maxiter: int = 1000*) → *NashComputationResult*

Compute approximate agent Nash equilibria of a game using *Lyapunov function minimization*.

Added in version 16.5.0: Moved from *liap_solve* passing a *MixedBehaviorProfileDouble* for additional clarity in the solution concept computed.

Parameters

- **start** (*MixedBehaviorProfileDouble*) – The starting profile for function minimization. Up to one equilibrium will be found from any starting profile, and the equilibrium found may (and generally will) depend on the initial profile chosen.
- **maxregret** (*float, default 1e-4*) – The acceptance criterion for approximate Nash equilibrium; the maximum regret of any player must be no more than *maxregret* times the difference of the maximum and minimum payoffs of the game
- **maxiter** (*int, default 1000*) – Maximum number of iterations in function minimization.

Returns
res – The result represented as a `NashComputationResult` object.

Return type
NashComputationResult

pygambit.nash.logit_solve

pygambit.nash.**logit_solve**(*game: Game*, *use_strategic: bool = False*, *maxregret: float = 1e-08*, *first_step: float = 0.03*, *max_accel: float = 1.1*) → *NashComputationResult*

Compute Nash equilibria of a game using *the logit quantal response equilibrium correspondence*.

Returns an approximation to the limiting point on the principal branch of the correspondence for the game.

Parameters

- **game** (*Game*) – The game to compute equilibria in.
- **use_strategic** (*bool, default False*) – Whether to use the strategic form. If True, always uses the strategic representation even if the game's native representation is extensive.
- **maxregret** (*float, default 1e-8*) – The acceptance criterion for approximate Nash equilibrium; the maximum regret of any player must be no more than *maxregret* times the difference of the maximum and minimum payoffs of the game

  Added in version 16.2.0.
- **first_step** (*float, default .03*) – The arclength of the initial step.

  Added in version 16.2.0.
- **max_accel** (*float, default 1.1*) – The maximum rate at which to lengthen the arclength step size.

  Added in version 16.2.0.

Returns
res – The result represented as a `NashComputationResult` object.

Return type
NashComputationResult

pygambit.nash.simpdiv_solve

pygambit.nash.**simpdiv_solve**(*start: MixedStrategyProfileRational*, *maxregret: Rational | None = None*, *refine: int = 2*, *leash: int | None = None*) → *NashComputationResult*

Compute Nash equilibria of a game using *simplicial subdivision*.

Changed in version 16.2.0: Method now takes a starting point, as a mixed strategy profile, instead of a game.

Parameters

- **start** (*MixedStrategyProfileRational*) – The starting profile for the algorithm. Up to one equilibrium will be found from any starting profile, and the equilibrium found may (and generally will) depend on the initial profile chosen.
- **maxregret** (*Rational, default 1e-8*) – The acceptance criterion for approximate Nash equilibrium; the maximum regret of any player must be no more than *maxregret* times the difference of the maximum and minimum payoffs of the game
- **refine** (*int, default 2*) – This controls the rate at which the triangulation of the space of mixed strategy profiles is made more fine at each iteration.

- **leash** (*int, optional*) – Simplicial subdivision is guaranteed to converge to an (approximate) Nash equilibrium. The method may take arbitrarily long paths through the space of mixed strategies in doing so. If specified, *leash* sets a maximum number of grid steps the method may explore. This trades off the possibility of finding an equilibrium more quickly by giving up the guarantee than an equilibrium will necessarily be found.

Returns
res – The result represented as a `NashComputationResult` object.

Return type
NashComputationResult

pygambit.nash.ipa_solve

pygambit.nash.**ipa_solve**(*perturbation:* Game | *MixedStrategyProfileDouble*) → *NashComputationResult*

Compute Nash equilibria of a game using *iterated polymatrix approximation*.

Parameters
perturbation (*Game or MixedStrategyProfileDouble*) – The perturbation vector to apply to the game. If a `Game` is passed, the perturbation vector is set to be 1 for the first strategy for each player and 0 for all other strategies.

Changed in version 16.2.0: Allow selection of the perturbation vector

Raises
ValueError – If the perturbation vector does not have a unique maximizer for each player

Returns
res – The result represented as a `NashComputationResult` object.

Return type
NashComputationResult

pygambit.nash.gnm_solve

pygambit.nash.**gnm_solve**(*perturbation:* Game | *MixedStrategyProfileDouble*, *end_lambda: float = -10.0*, *steps: int = 100*, *local_newton_interval: int = 3*, *local_newton_maxits: int = 10*) → *NashComputationResult*

Compute Nash equilibria of a game using *a global Newton method*.

Parameters

- **perturbation** (*Game or MixedStrategyProfileDouble*) – The perturbation vector to apply to the game. If a `Game` is passed, the perturbation vector is set to be 1 for the first strategy for each player and 0 for all other strategies.

  Changed in version 16.2.0: Allow selection of the perturbation vector

- **end_lambda** (*float, default -10.0*) – The value of the perturbation magnitude lambda at which to terminate tracing. This must be a negative number. This sets the point at which the algorithm assumes no further equilibria will be found along this ray.

  Added in version 16.2.0.

- **steps** (*int, default 100*) – The number of steps to take within a support cell. Lqrger values trade off speed for security in tracing the path.

  Added in version 16.2.0.

- **local_newton_interval** (*int, default 3*) – The frequency to run a local Newton method step. This is a correction step that reduces accumulated errors in the path-following.

Added in version 16.2.0.

- **local_newton_maxits** (*int, default 10*) – The maximum number of iterations in a local Newton method step.

  Added in version 16.2.0.

Raises

ValueError – If the perturbation vector does not have a unique maximizer for each player, or arguments controlling the behavior of the numerical tracing are not valid.

Returns

res – The result represented as a `NashComputationResult` object.

Return type

NashComputationResult

Computation of quantal response equilibria

logit_solve_branch(game[, use_strategic, ...])	
logit_solve_lambda(game, lam[, ...])	
logit_estimate(data[, use_empirical, ...])	Use maximum likelihood estimation to find the logit quantal response equilibrium which best fits empirical frequencies of play.
LogitQREMixedStrategyFitResult(data, method, ...)	The result of fitting a QRE to a given probability distribution over strategies.
LogitQREMixedBehaviorFitResult(data, method, ...)	The result of fitting a QRE to a given probability distribution over actions.

pygambit.qre.logit_solve_branch

pygambit.qre.**logit_solve_branch**(*game:* Game, *use_strategic: bool = False*, *maxregret: float = 1e-08*, *first_step: float = 0.03*, *max_accel: float = 1.1*)

pygambit.qre.logit_solve_lambda

pygambit.qre.**logit_solve_lambda**(*game:* Game, *lam: float | list[float]*, *use_strategic: bool = False*, *first_step: float = 0.03*, *max_accel: float = 1.1*)

pygambit.qre.logit_estimate

pygambit.qre.**logit_estimate**(*data:* MixedStrategyProfile | MixedBehaviorProfile, *use_empirical: bool = False*, *local_max: bool = False*, *first_step: float = 0.03*, *max_accel: float = 1.1*) → *LogitQREMixedStrategyFitResult* | *LogitQREMixedBehaviorFitResult*

Use maximum likelihood estimation to find the logit quantal response equilibrium which best fits empirical frequencies of play.

Added in version 16.3.0.

Parameters

- **data** (`MixedStrategyProfile` *or* `MixedBehaviorProfile`) – The empirical distribution of play to which to fit the QRE. To obtain the correct resulting log-likelihood, these should be expressed as total counts of observations of each action rather than probabilities. If a MixedBehaviorProfile is specified, estimation is done using the agent QRE.

- **use_empirical** (*bool, default = False*) – If specified and True, use the empirical payoff approach for estimation. This replaces the payoff matrix of the game with an approximation as a collection of individual decision problems based on the empirical expected payoffs to strategies or actions. This is computationally much faster but in most cases produces estimates which deviate systematically from those obtained by computing the QRE correspondence of the game. See the discussion in[1] for more details.
- **local_max** (*bool, default False*) – The default behavior is to find the global maximiser along the principal branch. If this parameter is set to True, tracing stops at the first interior local maximiser found.

  Note

  This argument only has an effect when use_empirical is False.

- **first_step** (*float, default .03*) – The arclength of the initial step.

  Note

  This argument only has an effect when use_empirical is False.

- **max_accel** (*float, default 1.1*) – The maximum rate at which to lengthen the arclength step size.

  Note

  This argument only has an effect when use_empirical is False.

Returns

The result of the estimation represented as a `LogitQREMixedStrategyFitResult` or `LogitQREMixedBehaviorFitResult` object, as appropriate.

Return type

LogitQREMixedStrategyFitResult or *LogitQREMixedBehaviorFitResult*

References

pygambit.qre.LogitQREMixedStrategyFitResult

class pygambit.qre.**LogitQREMixedStrategyFitResult**(*data*, *method*, *lam*, *profile*, *log_like*)

The result of fitting a QRE to a given probability distribution over strategies.

[1] Bland, J. R. and Turocy, T. L., 2023. Quantal response equilibrium as a structural model for estimation: The missing manual. SSRN working paper 4425515.

See also

logit_estimate

Methods

—

Attributes

data	The empirical strategy frequencies used to estimate the QRE.
lam	The value of lambda corresponding to the QRE.
log_like	The log-likelihood of the data at the estimated QRE.
method	The method used to estimate the QRE; either "fixed-point" or "empirical".
profile	The mixed strategy profile corresponding to the QRE.

pygambit.qre.LogitQREMixedBehaviorFitResult

class pygambit.qre.**LogitQREMixedBehaviorFitResult**(*data*, *method*, *lam*, *profile*, *log_like*)

The result of fitting a QRE to a given probability distribution over actions.

See also

logit_estimate

Methods

—

Attributes

data	The empirical actions frequencies used to estimate the QRE.
lam	The value of lambda corresponding to the QRE.
log_like	The log-likelihood of the data at the estimated QRE.
method	The method used to estimate the QRE; either "fixed-point" or "empirical".
profile	The mixed behavior profile corresponding to the QRE.

Catalog of games

load(slug)	Load a game from the package catalog.
games([n_actions, n_contingencies, ...])	List games available in the package catalog.

pygambit.catalog.load

pygambit.catalog.**load**(*slug: str*) → *Game*

Load a game from the package catalog.

Parameters
slug (*str*) – The slug of the game to load.

Returns
The loaded game.

Return type
gbt.Game

Raises
FileNotFoundError – If the game does not exist in the catalog.

pygambit.catalog.games

pygambit.catalog.**games**(*n_actions: int | None = None*, *n_contingencies: int | None = None*, *n_infosets: int | None = None*, *is_const_sum: bool | None = None*, *is_perfect_recall: bool | None = None*, *is_tree: bool | None = None*, *min_payoff: float | None = None*, *max_payoff: float | None = None*, *n_nodes: int | None = None*, *n_outcomes: int | None = None*, *n_players: int | None = None*, *n_strategies: int | None = None*, *include_descriptions: bool = False*) → DataFrame

List games available in the package catalog.

Most arguments are treated as filters on the attributes of the Game objects.

Parameters

- **n_actions** (*int, optional*) – The number of actions in the game. Only extensive games are returned.
- **n_contingencies** (*int, optional*) – The number of contingencies in the game.
- **n_infosets** (*int, optional*) – The number of information sets in the game. Only extensive games are returned.
- **is_const_sum** (*bool, optional*) – Whether the game is constant-sum.
- **is_perfect_recall** (*bool, optional*) – Whether the game has perfect recall.
- **is_tree** (*bool, optional*) – Whether the game is an extensive game (a tree).
- **min_payoff** (*float, optional*) – The minimum payoff in the game. Games returned have *min_payoff* >= *value*.
- **max_payoff** (*float, optional*) – The maximum payoff in the game. Games returned have *max_payoff* <= *value*.
- **n_nodes** (*int, optional*) – The number of nodes in the game. Only extensive games are returned.
- **n_outcomes** (*int, optional*) – The number of outcomes in the game.
- **n_players** (*int, optional*) – The number of players in the game.
- **n_strategies** (*int, optional*) – The number of pure strategies in the game.
- **include_descriptions** (*bool, optional*) – Whether to include the description of each game in the returned DataFrame. Defaults to False.

Returns

A DataFrame with columns “Game” and “Title”, where “Game” is the slug to load the game. If *include_descriptions=True*, the DataFrame will also include a “Description” column.

Return type

pd.DataFrame

CHAPTER

FOUR

CLI

See installation instructions in the *Installing Gambit GUI & CLI tools* section.

Gambit provides command-line interfaces for each method for computing Nash equilibria. These are suitable for scripting or calling from other programs. This chapter describes the use of these programs. For a general overview of methods for computing equilibria, see the survey of [**?**].

The graphical interface also provides a frontend for calling these programs and evaluating their output. Direct use of the command-line programs is intended for advanced users and applications.

These programs take an extensive or strategic game file, which can be specified on the command line or piped via standard input, and output a list of equilibria computed. The default output format is to present equilibria computed as a list of comma-separated probabilities, preceded by the tag *NE*. For mixed strategy profiles, the probabilities are sorted lexicographically by player, then by strategy. For behavior strategy profiles, the probabilities are sorted by player, then information set, then action number, where the information sets for a player are sorted by the order in which they are encountered in a depth-first traversal of the game tree. Many programs take an option *-D*, which, if specified, instead prints a more verbose, human-friendly description of each strategy profile computed.

Many of the programs optionally output additional information about the operation of the algorithm. These outputs have other, program-specific tags, described in the individual program documentation.

4.1 gambit-enumpure

Enumerate pure-strategy equilibria of a game. See the *algorithm description* for full details.

Changed in version 14.0.2: The effect of the *-S* switch is now purely cosmetic, determining how the equilibria computed are represented in the output. Previously, *-S* computed using the strategic game; if this was not specified for an extensive game, the agent form equilibria were returned.

-S

Report equilibria in reduced strategic form strategies, even if the game is an extensive game. By default, if passed an extensive game, the output will be in behavior strategies. Specifying this switch does not imply any change in operation internally, as pure-strategy equilibria are defined in terms of reduced strategic form strategies.

-D

Added in version 14.0.2.

The default output format for computed equilibria is a comma-separated list of strategy or action probabilities, suitable for postprocessing by automated tools. Specifying *-D* instead causes the program to output greater detail on each equilbrium profile computed.

-A

Added in version 14.0.2.

Report agent Nash equilibria, that is, equilibria which consider only deviations at a single information set at a time. Only has an effect for extensive games, as strategic games have only one information set per player.

-h

Prints a help message listing the available options.

-q

Suppresses printing of the banner at program launch.

Computing the pure-strategy equilibria of extensive game `e02.efg`, the example in Figure 2 of Selten (International Journal of Game Theory, 1975)

$ gambit-enumpure e02.efg

Search for Nash equilibria in pure strategies

Gambit version 16.6.0, Copyright (C) 1994-2026, The Gambit Project

This is free software, distributed under the GNU GPL

NE,1,0,0,0,1,0

With the *-S* switch, the set of equilibria returned is the same, except expressed in strategic game strategies rather than behavior strategies

$ gambit-enumpure -S e02.efg

Search for Nash equilibria in pure strategies Gambit version 16.6.0, Copyright (C) 1994-2026, The Gambit Project This is free software, distributed under the GNU GPL

NE,1,0,0,1,0

The *-A* switch considers only behavior strategy profiles where there is no way for a player to improve his payoff by changing action at only one information set; therefore the set of solutions is larger

$ gambit-enumpure -A e02.efg Search for Nash equilibria in pure strategies Gambit version 16.6.0, Copyright (C) 1994-2026, The Gambit Project This is free software, distributed under the GNU GPL

NE,1,0,1,0,1,0 NE,1,0,1,0,0,1 NE,1,0,0,1,1,0

4.2 gambit-enummixed

Enumerate equilibria in a two-player game. See the *algorithm description* for full details.

-d

By default, this program computes using exact rational arithmetic. Since the extreme points computed by this method are guaranteed to be rational when the payoffs in the game are rational, this permits exact computation of the equilibrium set. Computation using rational arithmetic is in general slow, however. For most games, acceptable results can be obtained by computing using the computer's native floating-point arithmetic. Using this flag enables computation in floating-point, and expresses all output using decimal representations with the specified number of digits.

-D

Since all Nash equilibria involve only strategies which survive iterative elimination of strictly dominated strategies, the program carries out the elimination automatically prior to computation. This is recommended, since it almost always results in superior performance. Specifying *-D* skips the elimination step and performs the enumeration on the full game.

-c

The program outputs the extreme equilibria as it finds them, prefixed by the tag NE . If this option is specified, once all extreme equilbria are identified, the program computes the convex sets which make up the set of equilibria. The program then additionally outputs each convex set, prefixed by convex-N , where N indexes the set. The set of all equilibria, then, is the union of these convex sets.

-h

Prints a help message listing the available options.

-q

Suppresses printing of the banner at program launch.

-L

Use lrslib by David Avis to carry out the enumeration process. This is an experimental feature that has not been widely tested.

Computing the equilibria, in mixed strategies, of `e02.nfg`, the reduced strategic form of the example in Figure 2 of Selten (International Journal of Game Theory, 1975)

> $ gambit-enummixed e02.nfg Compute Nash equilibria by enumerating extreme points Gambit version 16.6.0, Copyright (C) 1994-2026, The Gambit Project This is free software, distributed under the GNU GPL
>
> NE,1,0,0,1,0 NE,1,0,0,1/2,1/2

In fact, the game e02.nfg has a one-dimensional continuum of equilibria. This fact can be observed by examining the connectedness information using the *-c* switch

> $ gambit-enummixed -c e02.nfg Compute Nash equilibria by enumerating extreme points Gambit version 16.6.0, Copyright (C) 1994-2026, The Gambit Project This is free software, distributed under the GNU GPL
>
> NE,1,0,0,1,0 NE,1,0,0,1/2,1/2 convex-1,1,0,0,1/2,1/2 convex-1,1,0,0,1,0

4.3 gambit-enumpoly

Compute equilibria of a game using polynomial systems of equations See the *algorithm description* for full details.

When the verbose switch *-v* is used, the program outputs each support as it is considered. The supports are presented as a comma-separated list of binary strings, where each entry represents one player. The digit 1 represents a strategy which is present in the support, and the digit 0 represents a strategy which is not present. Each candidate support is printed with the label "candidate,".

The approach of subdividing the space of totally mixed profiles assumes solutions to the system of equations and inequalities are isolated points. In the case of degeneracies in the resulting system, When the verbose switch *-v* is used, these supports are identified on standard output with the label "singular,". This will occur if there is a positive-dimensional set of equilibria which all share the listed support. However, the converse is not true: not all supports labeled as "singular" will necessarily be the support of some set of equilibria.

-d

Express all output using decimal representations with the specified number of digits.

-h

Prints a help message listing the available options.

-H

By default, the program uses an enumeration method designed to visit as few supports as possible in searching for all equilibria. With this switch, This switch only has an effect when solving strategic games.

-S

By default, the program uses behavior strategies for extensive games; this switch instructs the program to use reduced strategic game strategies for extensive games. (This has no effect for strategic games, since a strategic game is its own reduced strategic game.)

-m

Added in version 16.3.0.

Specify the maximum regret criterion for acceptance as an approximate Nash equilibrium (default is 1e-4). See pygambit-nash-maxregret for interpretation and guidance.

-e EQA

Added in version 16.3.0.

By default, the program will search all support profiles. This switch instructs the program to terminate when EQA equilibria have been found.

-q

Suppresses printing of the banner at program launch.

-v

Sets verbose mode. In verbose mode, supports are printed on standard output with the label "candidate" as they are considered, and singular supports are identified with the label "singular." By default, no information about supports is printed.

Computing equilibria of the strategic game `e01.nfg`, the example in Figure 1 of Selten (International Journal of Game Theory, 1975) sometimes called "Selten's horse"

$ gambit-enumpoly e01.nfg Compute Nash equilibria by solving polynomial systems Gambit version 16.6.0, Copyright (C) 1994-2026, The Gambit Project This is free software, distributed under the GNU GPL

NE,1.000000,0.000000,1.000000,0.000000,0.000000,1.000000 NE,0.000000,1.000000,1.000000,0.000000,1.000000,0.000000
NE,0.000000,1.000000,0.333333,0.666667,1.000000,0.000000 NE,1.000000,0.000000,1.000000,0.000000,0.250000,0.750000

4.4 gambit-lcp

Compute equilibria in a two-player game via linear complementarity. See the *algorithm description* for full details.

-d

By default, this program computes using exact rational arithmetic. Since the extreme points computed by this method are guaranteed to be rational when the payoffs in the game are rational, this permits exact computation of the equilibrium set. Computation using rational arithmetic is in general slow, however. For most games, acceptable results can be obtained by computing using the computer's native floating-point arithmetic. Using this flag enables computation in floating-point, and expresses all output using decimal representations with the specified number of digits.

-S

By default, the program uses behavior strategies for extensive games; this switch instructs the program to use reduced strategic game strategies for extensive games. (This has no effect for strategic games, since a strategic game is its own reduced strategic game.)

-D

Added in version 14.0.2.

The default output format for computed equilibria is a comma-separated list of strategy or action probabilities, suitable for postprocessing by automated tools. Specifying *-D* instead causes the program to output greater detail on each equilbrium profile computed.

-e EQA

By default, when working with the reduced strategic game, the program will find all equilibria accessible from the origin of the polytopes. This switch instructs the program to terminate when EQA equilibria have been

found. This has no effect when using the extensive representation of a game, in which case the method always only returns one equilibrium.

-h

Prints a help message listing the available options.

-q

Suppresses printing of the banner at program launch.

Computing an equilibrium of extensive game `e02.efg`, the example in Figure 2 of Selten (International Journal of Game Theory, 1975)

$ gambit-lcp e02.efg Compute Nash equilibria by solving a linear complementarity program Gambit version 16.6.0, Copyright (C) 1994-2026, The Gambit Project This is free software, distributed under the GNU GPL

NE,1,0,1/2,1/2,1/2,1/2

4.5 gambit-lp

Compute equilibria in a two-player constant-sum game via linear programming. See the *algorithm description* for full details.

-d

By default, this program computes using exact rational arithmetic. Since the extreme points computed by this method are guaranteed to be rational when the payoffs in the game are rational, this permits exact computation of an equilibrium. Computation using rational arithmetic is in general slow, however. For most games, acceptable results can be obtained by computing using the computer's native floating-point arithmetic. Using this flag enables computation in floating-point, and expresses all output using decimal representations with the specified number of digits.

-S

By default, the program uses behavior strategies for extensive games; this switch instructs the program to use reduced strategic game strategies for extensive games. (This has no effect for strategic games, since a strategic game is its own reduced strategic game.)

-D

Added in version 14.0.3.

The default output format for computed equilibria is a comma-separated list of strategy or action probabilities, suitable for postprocessing by automated tools. Specifying *-D* instead causes the program to output greater detail on each equilbrium profile computed.

-h

Prints a help message listing the available options.

-q

Suppresses printing of the banner at program launch.

Computing an equilibrium of the game `2x2const.nfg`, a game with two players with two strategies each, with a unique equilibrium in mixed strategies

$ gambit-lp 2x2const.nfg Compute Nash equilibria by solving a linear program Gambit version 16.6.0, Copyright (C) 1994-2026, The Gambit Project This is free software, distributed under the GNU GPL

NE,1/3,2/3,1/3,2/3

4.6 gambit-liap

Compute Nash equilibria using function minimization. See the *algorithm description* for full details.

Changed in version 16.2.0: The Lyapunov function is now normalized to be independent of the scale of the payoffs of the game; therefore multiplying or dividing all payoffs by a common factor will not affect the output of the algorithm.

The criterion for accepting whether a local constrained minimizer of the Lyanunov function is an approximate Nash equilibrium is specified in terms of the maximum regret. This regret is interpreted as a fraction of the difference between the maximum and minimum payoffs in the game.

Changed in version 16.5.0: The *-A* switch has been introduced to be explicit in choosing to compute agent Nash equilibria. The default is now to compute using the strategic form even for extensive games.

-A

Added in version 16.5.0.

Report agent Nash equilibria, that is, equilibria which consider only deviations at a single information set at a time. Only has an effect for extensive games, as strategic games have only one information set per player.

-d

Express all output using decimal representations with the specified number of digits.

-n

Specify the number of starting points to randomly generate.

-i

Added in version 16.1.1.

Specify the maximum number of iterations in function minimization (default is 1000).

-m

Added in version 16.2.0.

Specify the maximum regret criterion for acceptance as an approximate Nash equilibrium (default is 1e-4). See pygambit-nash-maxregret for interpretation and guidance.

-h

Prints a help message listing the available options.

-q

Suppresses printing of the banner at program launch.

-s

Specifies a file containing a list of starting points for the algorithm. The format of the file is comma-separated values, one mixed strategy profile per line, in the same format used for output of equilibria (excluding the initial NE tag).

-S

By default, the program uses behavior strategies for extensive games; this switch instructs the program to use reduced strategic game strategies for extensive games. (This has no effect for strategic games, since a strategic game is its own reduced strategic game.)

-v

Sets verbose mode. In verbose mode, initial points, as well as points at which the minimization fails at a constrained local minimum that is not a Nash equilibrium, are all output, in addition to any equilibria found.

Computing an equilibrium in mixed strategies of `e02.efg`, the example in Figure 2 of Selten (International Journal of Game Theory, 1975)

$ gambit-liap e02.nfg Compute Nash equilibria by minimizing the Lyapunov function Gambit version 16.6.0, Copyright (C) 1994-2026, The Gambit Project This is free software, distributed under the GNU GPL

NE,0.998701,0.000229,0.001070,0.618833,0.381167

4.7 gambit-simpdiv

Compute equilibria via simplicial subdivision. See the *algorithm description*.

The algorithm begins with any mixed strategy profile consisting of rational numbers as probabilities. Without any options, the algorithm begins with the centroid, and computes one Nash equilibrium. To attempt to compute other equilibria that may exist, use the `gambit-simpdiv -r` or `gambit-simpdiv -s` options to specify additional starting points for the algorithm.

-g

Sets the granularity of the grid refinement. By default, when the grid is refined, the stepsize is cut in half, which corresponds to specifying *-g 2*. If this parameter is specified, the grid is refined at each step by a multiple of MULT .

-h

Prints a help message listing the available options.

-n

Randomly generate COUNT starting points. Only applicable if option `gambit-simpdiv -r` is also specified.

-q

Suppresses printing of the banner at program launch.

-r

Generate random starting points with denominator DENOM. Since this algorithm operates on a grid, by its nature the probabilities it works with are always rational numbers. If this parameter is specified, starting points for the procedure are generated randomly using the uniform distribution over strategy profiles with probabilities having denominator DENOM.

-s

Specifies a file containing a list of starting points for the algorithm. The format of the file is comma-separated values, one mixed strategy profile per line, in the same format used for output of equilibria (excluding the initial NE tag).

-m

Added in version 16.2.0.

Specify the maximum regret criterion for acceptance as an approximate Nash equilibrium (default is 1e-8). See pygambit-nash-maxregret for interpretation and guidance.

-d DECIMALS

Added in version 16.2.0.

Simplicial subdivision operates on a triangulation grid in the set of mixed strategy profiles. Therefore, it produces output in which all probabilities are expressed as rational numbers, and by default the output reports these. By specifying this option, instead probabilities are expressed as floating-point numbers with the specified number of decimal places. Specifying this option sacrifices some precision in reporting the output of the method, in exchange for probabilities which are more human-readable.

-v

Sets verbose mode. In verbose mode, initial points, as well as the approximations computed at each grid refinement, are all output, in addition to the approximate equilibrium profile found.

Computing an equilibrium in mixed strategies of `e02.efg`, the example in Figure 2 of Selten (International Journal of Game Theory, 1975)

> $ gambit-simpdiv e02.nfg Compute Nash equilibria using simplicial subdivision Gambit version 16.6.0, Copyright (C) 1994-2026, The Gambit Project This is free software, distributed under the GNU GPL
>
> NE,1,0,0,1,0

4.8 gambit-logit

Compute and/or estimate quantal response equilbria. See the *algorithm description* for full details.

Changed in version 16.2.0: The criterion for accepting whether a point is sufficiently close to a Nash equilibrium to terminate the path-following is specified in terms of the maximum regret. This regret is interpreted as a fraction of the difference between the maximum and minimum payoffs in the game.

-d

Express all output using decimal representations with the specified number of digits. The default is *-d 6*.

-s

Sets the initial step size for the predictor phase of the tracing procedure. The default value is .03. The step size is specified in terms of the arclength along the branch of the correspondence, and not the size of the step measured in terms of lambda. So, for example, if the step size is currently .03, but the position of the strategy profile on the branch is changing rapidly with lambda, then lambda will change by much less then .03 between points reported by the program.

-a

Sets the maximum acceleration of the step size during the tracing procedure. This is interpreted as a multiplier. The default is 1.1, which means the step size is increased or decreased by no more than ten percent of its current value at every step. A value close to one would keep the step size (almost) constant at every step.

-m

Added in version 16.2.0.

Specify the maximum regret criterion for acceptance as an approximate Nash equilibrium (default is 1e-8). See pygambit-nash-maxregret for interpretation and guidance.

-l

While tracing, compute the logit equilibrium points with parameter LAMBDA accurately. This option may be specified multiple times, in which case points are found for each successive lambda, in the order specified, along the branch.

Changed in version 16.3.0: Added support for specifying multiple lambda values.

-S

By default, the program uses behavior strategies for extensive games; this switch instructs the program to use reduced strategic game strategies for extensive games. (This has no effect for strategic games, since a strategic game is its own reduced strategic game.)

-h

Prints a help message listing the available options.

-e

By default, all points computed are output by the program. If this switch is specified, only the approximation to the Nash equilibrium at the end of the branch is output.

Computing the principal branch, in mixed strategies, of `e02.nfg`, the reduced strategic form of the example in Figure 2 of Selten (International Journal of Game Theory, 1975)

$ gambit-logit e02.nfg Compute a branch of the logit equilibrium correspondence Gambit version 16.6.0, Copyright (C) 1994-2026, The Gambit Project This is free software, distributed under the GNU GPL

0.000000,0.333333,0.333333,0.333333,0.5,0.5 0.022853,0.335873,0.328284,0.335843,0.501962,0.498038
0.047978,0.338668,0.322803,0.33853,0.504249,0.495751 0.075600,0.341747,0.316863,0.34139,0.506915,0.493085
0.105965,0.345145,0.310443,0.344413,0.510023,0.489977 0.139346,0.348902,0.303519,0.347578,0.51364,0.48636

...

735614.794714,1,0,4.40659e-11,0.500016,0.499984 809176.283787,1,0,3.66976e-11,0.500015,0.499985 890093.921767,1,0,3.05596e-11,0.500014,0.499986
979103.323545,1,0,2.54469e-11,0.500012,0.499988 1077013.665501,1,0,2.11883e-11,0.500011,0.499989

4.9 gambit-gnm

Compute Nash equilibria in a strategic game using a global Newton method. See the *algorithm description* for full details.

The algorithm finds a subset of equilibria starting from any given profile. Multiple starting profiles may be generated via the *-n* option or specified via the *-s* option; different starting profiles may result in different subsets of equilibria being found.

-d

Express all output using decimal representations with the specified number of digits.

-h

Prints a help message listing the available options.

-n

Randomly generate the specified number of perturbation vectors.

-q

Suppresses printing of the banner at program launch.

-s

Specifies a file containing a list of starting points for the algorithm. The format of the file is comma-separated values, one mixed strategy profile per line, in the same format used for output of equilibria (excluding the initial NE tag).

-m LAMBDA

Added in version 16.2.0.

Specifies the value of lambda at which to assume no more equilibria are accessible via the specified ray, and terminate tracing. Must be a negative number; default is -10.

-f FREQ

Added in version 16.2.0.

Specifies the frequency to run a local Newton method step. This is a correction step that reduces accumulated errors in the path-following. Default is 3.

-i MAXITS

Added in version 16.2.0.

Specifies the maximum number of iterations in a local Newton method step. Default is 10.

-c STEPS

Added in version 16.2.0.

Specifies the number of steps to take within a support cell. Larger values trade off speed for security in tracing the path. Default is 100.

-v

Show intermediate output of the algorithm. If this option is not specified, only the equilibria found are reported.

Computing an equilibrium of `e02.nfg`, the reduced strategic form of the example in Figure 2 of Selten (International Journal of Game Theory, 1975)

$ gambit-gnm e02.nfg Compute Nash equilibria using a global Newton method Gametracer version 0.2, Copyright (C) 2002, Ben Blum and Christian Shelton Gambit version 16.6.0, Copyright (C) 1994-2026, The Gambit Project This is free software, distributed under the GNU GPL NE,1,0,2.99905e-12,0.5,0.5

See also

gambit-ipa.

4.10 gambit-ipa

Compute Nash equilibria in a strategic game using iterated polymatrix approximation. See the *algorithm description* for full details.

The algorithm finds at most one equilibrium starting from any given profile. Multiple starting profiles may be generated via the *-n* option or specified via the *-s* option; different starting profiles may result in different equilibria being found.

-d

Express all output using decimal representations with the specified number of digits.

-h

Prints a help message listing the available options.

-n

Randomly generate the specified number of perturbation vectors.

-q

Suppresses printing of the banner at program launch.

-s

Specifies a file containing a list of starting points for the algorithm. The format of the file is comma-separated values, one mixed strategy profile per line, in the same format used for output of equilibria (excluding the initial NE tag).

Computing an equilibrium of `e02.nfg`, the reduced strategic form of the example in Figure 2 of Selten (International Journal of Game Theory, 1975)

$ gambit-ipa e02.nfg Compute Nash equilibria using iterated polymatrix approximation Gametracer version 0.2, Copyright (C) 2002, Ben Blum and Christian Shelton Gambit version 16.6.0, Copyright (C) 1994-2026, The Gambit Project This is free software, distributed under the GNU GPL

NE,1.000000,0.000000,0.000000,1.000000,0.000000

See also

gambit-gnm.

4.11 gambit-convert

gambit-convert reads a game on standard input in any supported format and converts it to another text representation. Currently, this tool supports outputting the strategic form of the game in one of these formats:

- A standard HTML table.
- A LaTeX fragment in the format of Martin Osborne's *sgame* macros (see http://www.economics.utoronto.ca/osborne/latex/index.html).

-O FORMAT

Required. Specifies the output format. Supported options for *FORMAT* are *html* or *sgame*.

-r PLAYER

Specifies the player number to place on the rows of the tables. The default if not specified is to place player 1 on the rows.

-c PLAYER

Specifies the player number to place on the columns of the tables. The default if not specified is to place player 2 on the columns.

-h

Prints a help message listing the available options.

-q

Suppresses printing of the banner at program launch.

Example invocation for HTML output

$ gambit-convert -O html 2x2.nfg Convert games among various file formats Gambit version 16.6.0, Copyright (C) 1994-2026, The Gambit Project This is free software, distributed under the GNU GPL

<h1>Two person 2 x 2 game with unique mixed equilibrium</h1> <table><tr><td></td><td align=center>1</td><td align=center>2</td></tr><tr><td align=center>1</td><td align=center>2,0</td><td align=center>0,1</td></tr><tr><td align=center>2</td><td align=center>0,1</td><td align=center>1,0</td></tr></table>

Example invocation for LaTeX output

$ gambit-convert -O sgame 2x2.nfg Convert games among various file formats Gambit version 16.6.0, Copyright (C) 1994-2026, The Gambit Project This is free software, distributed under the GNU GPL

begin{game}{2}{2}[Player 1][Player 2] &1 & 2\ 1 & $2,0$ & $0,1$ \ 2 & $0,1$ & $1,0$ end{game}

CHAPTER

FIVE

GUI

See installation instructions in the *Installing Gambit GUI & CLI tools* section.

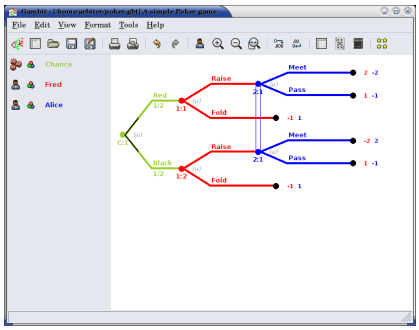
Gambit's graphical user interface provides an "integrated development environment" to help visually construct games and to investigate their main strategic features.

The graphical interface is largely intended for the interactive construction and analysis of small to medium games. Repeating the caution from the introduction of this manual, the computation time required for the equilibrium analysis of games increases rapidly in the size of the game. The graphical interface is ideal for students learning about the fundamentals of game theory, or for practitioners prototyping games of interest.

In graduating to larger applications, users are encouraged to make use of the underlying Gambit libraries and programs directly. For greater control over computing Nash and quantal response equilibria of a game, see the section on *the command-line tools*. To build larger games or to explore parameter spaces of a game systematically, it is recommended to use *the Python package*.

5.1 General concepts

5.1.1 General layout of the main window



The frame presenting a game consists of two principal panels. The main panel, to the right, displays the game graphically; in this case, showing the game tree of a simple one-card poker game. To the left is the player panel, which lists the players in the game; here, Fred and Alice are the players. Note that where applicable, information is color-coded to match the colors assigned to the players: Fred's moves and payoffs are all presented in red, and Alice's in blue. The color assigned to a player can be changed by clicking on the color icon located to the left of the player's name on the player panel. Player names are edited by clicking on the player's name, and editing the name in the text control that appears.

Two additional panels are available. Selecting *Tools* → *Dominance* toggles the display of an additional toolbar across the top of the window. This toolbar controls the indication and elimination of actions or strategies that are dominated. The use of this toolbar is discussed in *Investigating dominated strategies*.

Selecting *View* → *Profiles*, or clicking the show profiles icon on the toolbar, toggles the display of the list of computed strategy profiles on the game. More on the way the interface handles the computation of Nash equilibria and other kinds of strategy profiles is presented in *Computing Nash equilibria*.

5.1.2 Payoffs and probabilities in Gambit

Gambit stores all payoffs in games in an arbitrary-precision format. Payoffs may be entered as decimal numbers with arbitrarily many decimal places. In addition, Gambit supports representing payoffs using rational numbers. So, for example, in any place in which a payoff may appear, either an outcome of an extensive game or a payoff entry in a strategic game, the payoff one-tenth may be entered either as .1 or 1/10.

The advantage of this format is that, in certain circumstances, Gambit may be able to compute equilibria exactly. In addition, some methods for computing equilibria construct good numerical approximations to equilibrium points. For these methods, the computed equilibria are stored in floating-point format. To increase the number of decimal places shown for these profiles, click the increase decimals icon . To decrease the number of decimal places shown, click the decrease decimals icon .

Increasing or decreasing the number of decimals displayed in computed strategy profiles will not have any effect on the display of outcome payoffs in the game itself, since those are stored in arbitrary precision.

5.1.3 A word about file formats

The graphical interface manipulates several different file formats for representing games. This section gives a quick overview of those formats.

Gambit has for many years supported two file formats for representing games, one for extensive games (typically using the filename extension .efg) and one for strategic games (typically using the filename extension .nfg). These file formats are recognized by all Gambit versions dating back to release 0.94 in 1995. (Users interested in the details of these file formats can consult *Game representation formats* for more information.)

Beginning with release 2005.12.xx, the graphical interface now reads and writes a new file format, which is referred to as a"Gambit workbook." This extended file format stores not only the representation of the game, but also additional information, including parameters for laying out the game tree, the colors assigned to players, any equilibria or other analysis done on the game, and so forth. So, for example, the workbook file can be used to store the analysis of a game and then return to it. These files by convention end in the extension .gbt.

The graphical interface will read files in all three formats: .gbt, .efg, and .nfg. The "Save" and "Save as" commands, however, always save in the Gambit workbook (.gbt) format. To save the game itself as an extensive (.efg) or strategic (.nfg) game, use the items on the "Export" submenu of the "File" menu. This is useful in interfacing with older versions of Gambit, with other tools which read and write those formats, and in using the underlying Gambit analysis command-line tools directly, as those programs accept .efg or .nfg game files. Users primarily interested in using Gambit solely via the graphical interface are encouraged to use the workbook (.gbt) format.

As it is a new format, the Gambit workbook format is still under development and may change in details. It is intended that newer versions of the graphical interface will still be able to read workbook files written in older formats.

5.2 Extensive games

The graphical interface provides a flexible set of operations for constructing and editing general extensive games. These are outlined below.

5.2.1 Creating a new extensive game

To create a new extensive game, select *File* → *New* → *Extensive game*, or click on the new extensive game icon . The extensive game created is a trivial game with two players, named by default *Player 1* and *Player 2*, with one node, which is both the root and terminal node of the game. In addition, extensive games have a special player labeled *Chance*, which is used to represent random events not controlled by any of the strategic players in the game.

5.2.2 Adding moves

There are two options for adding moves to a tree: drag-and-drop and the *Insert move* dialog.

1. Moves can be added to the tree using a drag-and-drop idiom. From the player list window, drag the player icon located to the left of the player who will have the move to any terminal node in the game tree. The tree will be extended with a new move for that player, with two actions at the new move. Adding a move for the chance player is done the same way, except the dice icon appearing to the left of the chance player in the player list window is used instead of the player icon. For the chance player, the two actions created will each be given a probability weight of one-half. If the desired move has more than two actions, additional actions can be added by dragging the same player's icon to the move node; this will add one action to the move each time this is done.

2. Click on any terminal node in the tree, and select *Edit* → *Insert move* to display the *insert move* dialog. The dialog is intended to read like a sentence:

   - The first control specifies the player who will make the move. The move can be assigned to a new player by specifying *Insert move for a new player here*.
   - The second control selects the information set to which to add the move. To create the move in a new information set, select *at a new information set* for this control.
   - The third control sets the number of actions. This control is disabled unless the second control is set to *at a new information set*. Otherwise, it is set automatically to the number of actions at the selected information set.

The two methods can be useful in different contexts. The drag-and-drop approach is a bit quicker to use, especially when creating trees that have few actions at each move. The dialog approach is a bit more flexible, in that a move can be added for a new, as-yet-undefined player, a move can be added directly into an existing information set, and a move can be immediately given more than two actions.

5.2.3 Copying and moving subtrees

Many extensive games have structures that appear in multiple parts of the tree. It is often efficient to create the structure once, and then copy it as needed elsewhere.

Gambit provides a convenient idiom for this. Clicking on any nonterminal node and dragging to any terminal node implements a move operation, which moves the entire subtree rooted at the original, nonterminal node to the terminal node.

To turn the operation into a copy operation:

- On Windows and Linux systems, hold down the `Ctrl` key during the operation.
- On OS X, hold down the `Cmd` key when starting the drag operation, then release prior to dropping.

The entire subtree rooted at the original node is copied, starting at the terminal node. In this copy operation, each node in the copied image is placed in the same information set as the corresponding node in the original subtree.

Copying a subtree to a terminal node in that subtree is also supported. In this case, the copying operation is halted when reaching the terminal node, to avoid an infinite loop. Thus, this feature can also be helpful in constructing multiple-stage games.

5.2.4 Removing parts of a game tree

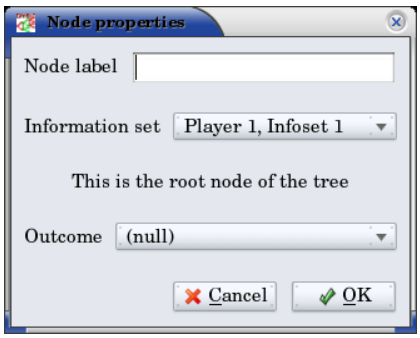
Two deletion operations are supported on extensive games. To delete the entire subtree rooted at a node, click on that node and select *Edit* → *Delete subtree*.

To delete an individual move from the game, click on one of the direct children of that node, and select *Edit* → *Delete parent*. This operation deletes the parent node, as well as all the children of the parent other than the selected node. The selected child node now takes the place of the parent node in the tree.

5.2.5 Managing information sets

Gambit provides several methods to help manage the information structure in an extensive game.

When building a tree, new moves can be placed in a given information set using the *Insert move dialog*. Additionally, new moves can be created using the drag-and-drop idiom by holding down the `Shift` key and dragging a node in the tree. During the drag operation, the cursor changes to the move icon . Dropping the move icon on another node places the target node in the same information set as the node where the drag operation began.



The information set to which a node belongs can also be set by selecting *Edit* → *Node*. This displays the *node properties* dialog. The *Information set* dropdown defaults to the current information set to which the node belongs, and contains a list of all other information sets in the game which are compatible with the node, that is, which have the same number of actions. Additionally, the node can be moved to a new, singleton information set by setting this dropdown to the *New information set* entry.

When building out a game tree using the *drag-and-drop approach* to copying portions of the tree, the nodes created in the copy of the subtree remain in the same information set as the corresponding nodes in the original subtree. In many cases, though, these trees differ in the information available to some or all of the players. To help speed the process of adjusting information sets in bulk, Gambit offers a "reveal" operation, which breaks information sets based on the action taken at a particular node. Click on a node at which the action taken is to be made known subsequently to other players, and select *Edit* → *Reveal*. This displays a dialog listing the players in the game. Check the boxes next to the players who observe the outcome of the move at the node, and click *OK*. The information sets at nodes below the selected one are adjusted based on the action selected at this node.

> **Note**
>
> The reveal operation only has an effect at the time it is done. In particular, it does not enforce the separation of information sets based on this information during subsequent editing of the game.

5.2.6 Outcomes and payoffs

Gambit supports the specification of payoffs at any node in a game tree, whether terminal or not. Each node is created with no outcome attached; in this case, the payoff at each node is zero to all players. These are indicated in the game tree by the presence of a *(u)* in light grey to the right of a node.

To set the payoffs at a node, double-click on the *(u)* to the right of the node. This creates a new outcome at the node, with payoffs of zero for all players, and displays an editor to set the payoff of the first player.

The payoff to a player for an outcome can be edited by double-clicking on the payoff entry. This action creates a text edit control in which the payoff to that player can be modified. Edits to the payoff can be accepted by pressing the `Enter` key. In addition, accepting the payoff by pressing the `Tab` key both stores the changes to the player's payoff, and advances the editor to the payoff for the next player at that outcome.

Outcomes may also be moved or copied using a drag-and-drop idiom. Left-clicking and dragging an outcome to another node moves the outcome from the original node to the target node. Copying an outcome may be accomplished by doing this same action while holding down the Control (`Ctrl`) key on the keyboard.

When using the copy idiom described above, the action assigns the same outcome to both the involved nodes. Therefore, if subsequently the payoffs of the outcome are edited, the payoffs at both nodes will be modified. To copy the outcome in such a way that the outcome at the target node is a different outcome from the one at the source, but with the same payoffs, hold down the `Shift` key instead of the `Control` key while dragging.

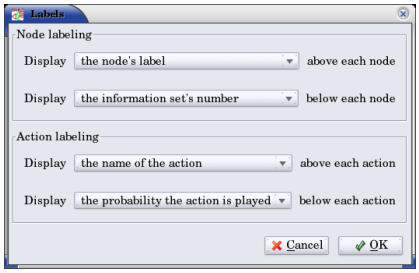
To remove an outcome from a node, click on the node, and select *Edit* → *Remove outcome*.

5.2.7 Formatting and labeling the tree

Gambit offers some options for customizing the display of game trees.

Labels on nodes and branches

The information displayed at the nodes and on the branches of the tree can be configured by selecting *Format* → *Labels*, which displays the *tree labels* dialog.



Above and below each node, the following information can be displayed:

No label
: The space is left blank.

The node's label
: The text label assigned to the node. (This is the default labeling above each node.)

The player's name
: The name of the player making the move at the node.

The information set's label
: The name of the information set to which the node belongs.

The information set's number
: A unique identifier of the information set, in the form player number:information set number. (This is the default labeling below each node.)

The realization probability
: The probability the node is reached. (Only displayed when a behavior strategy is selected to be displayed on the tree.)

The belief probability
: The probability a player assigns to being at the node, conditional on reaching the information set. (Only displayed when a behavior strategy is selected to be displayed on the tree.)

The payoff of reaching the node
: The expected payoff to the player making the choice at the node, conditional on reaching the node. (Only displayed when a behavior strategy is selected to be displayed on the tree.)

Above and below each branch, the following information can be displayed:

No label
: The space is left blank.

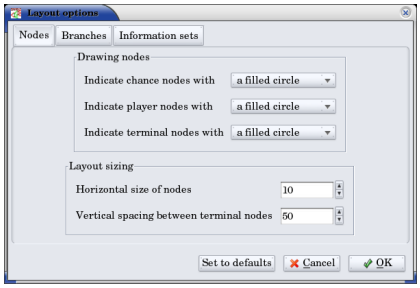
The name of the action
: The name of the action taken on the branch. (This it the default labeling above the branch.)

The probability the action is played
: For chance actions, the probability the branch is taken; this is always displayed. For player actions, the probability the action is taken in the selected profile (only displayed when a behavior strategy is selected to be displayed on the tree). In some cases, behavior strategies do not fully specify behavior sufficiently far off the equilibrium path; in such cases, an asterisk is shown for such action probabilities. (This is the default labeling below each branch.)

The value of the action
: The expected payoff to the player of taking the action, conditional on reaching the information set. (Only displayed when a behavior strategy is selected to be displayed on the tree.)

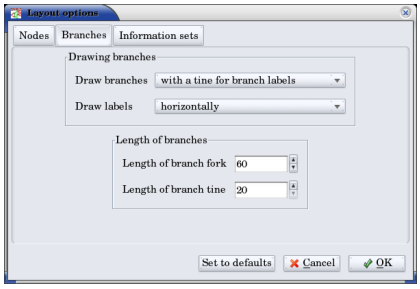
Controlling the layout of the tree

Gambit implements an automatic system for layout out game trees, which provides generally good results for most games. These can be adjusted by selecting *Format* → *Layout*. The layout parameters are organized on three tabs.



The first tab, labeled *Nodes*, controls the size, location, and rendering of nodes in the tree. Nodes can be indicated using one of five tokens: a horizontal line (the "traditional" Gambit style from previous versions), a box, a diamond, an unfilled circle, and a filled circle). These can be set independently to distinguish chance and terminal nodes from player nodes.

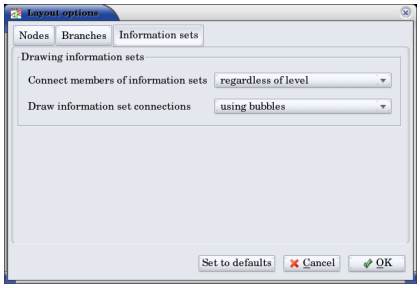
The sizing of nodes can be configured for best results. Gambit styling from previous versions used the horizontal line tokens with relatively long lines; when using the other tokens, smaller node sizes often look better.



The layout algorithm is based upon identifying the location of terminal nodes. The vertical spacing between these nodes can be set; making this value larger will tend to give the tree a larger vertical extent.

The second tab, labeled *Branches*, controls the display of the branches of the tree. The traditional Gambit way of drawing branches is a "fork-tine" approach, in which there is a flat part at the end of each branch at which labels are displayed. Alternatively, branches can be drawn directly between nodes by setting *Draw branches* to using straight lines between nodes. With this setting, labels are now displayed at points along the (usually) diagonal branches. Labels are usually shown horizontally; however, they can be drawn rotated parallel to the branches by setting *Draw labels* to rotated.

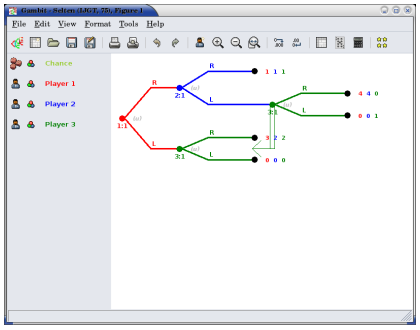
The rotated label drawing is experimental, and does not always look good on screen.



The length used for branches and their tines, if drawn, can be configured. Longer branch and tine lengths give more space for longer labels to be drawn, at the cost of giving the tree a larger horizontal extent.

Finally, display of the information sets in the game is configured under the tab labeled *Information sets*. Members of information sets are by default connected using a "bubble" similar to that drawn in textbook diagrams of games. The can be modified to use a single line to connect nodes in the same information set. In conjunction with using lines for nodes, this can sometimes lead to a more compact representation of a tree where there are many information sets at the same horizontal location.

The layout of the tree may be such that members of the same information set appear at different horizontal locations in the tree. In such a case, by default, Gambit draws a horizontal arrow pointing rightward or leftward to indicate the continuation of the information set, as illustrated in the diagram nearby.



These connections can be disabled by setting *Connect members of information sets* to *only when on the same level*. In addition, drawing information set indicators can be disabled entirely by setting this to invisibly (don't draw indicators).

Selecting fonts and colors

To select the font used to draw the labels in the tree, select *Format* → *Font*. The standard font selection dialog for the operating system is displayed, showing the fonts available on the system. Since available fonts vary across systems, when opening a workbook on a system different from the system on which it was saved, Gambit tries to match the font style as closely as possible when the original font is not available.

The color-coding for each player can be changed by clicking on the color icon to the left of the corresponding player.

5.3 Strategic games

Gambit has full support for constructing and manipulating arbitrary N-player strategic (also known as normal form) games.

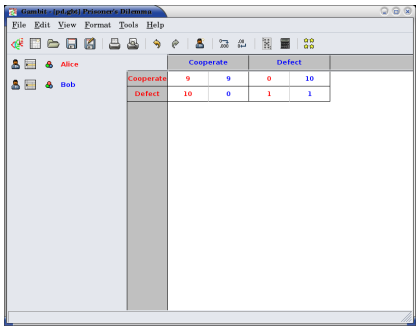
For extensive games, Gambit automatically computes the corresponding reduced strategic game. To view the reduced strategic game corresponding to an extensive game, select *View* → *Strategic game* or click the strategic game table icon on the toolbar.

The strategic games computed by Gambit as the reduced strategic game of an extensive game cannot be modified directly. Instead, edit the original extensive game; Gambit automatically recomputes the strategic game after any changes to the extensive game.

Strategic games may also be input directly. To create a new strategic game, select *File* → *New* → *Strategic game*, or click the new strategic game icon on the toolbar.

5.3.1 Navigating a strategic game

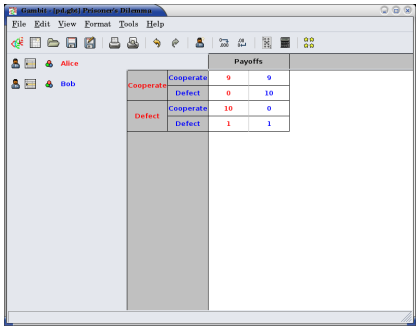
Gambit displays a strategic game in table form. All players are assigned to be either row players or column players, and the payoffs for each entry in the strategic game table correspond to the payoffs corresponding to the situation in which all the row players play the strategy specified on that row for them, and all the column players play the strategy specified on that column for them.



For games with two players, this presentation is by default configured to be similar to the standard presenation of strategic games as tables, in which one player is assigned to be the "row" player and the other the "column" player. However, Gambit permits a more flexible assignment, in which multiple players can be assigned to the rows and multiple players to the columns. This is of particular use for games with more than two players. In print, a three-player strategic game is usually presented as a collection of tables, with one player choosing the row, the second the column, and the third the table. Gambit presents such games by hierarchially listing the strategies of one or more players on both rows and columns.

The hierarchical presentation of the table is similar to that of a contingency table in a spreadsheet application. Here, Alice, shown in red, has her strategies listed on the rows of the table, and Bob, shown in blue, has his strategies listed on the columns of the table.

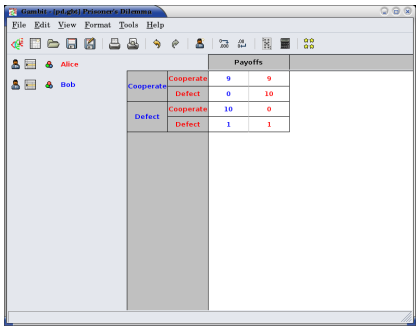
The assignment of players to row and column roles is fully customizable. To change the assignment of a player, drag the person icon appearing to the left of the player's name on the player toolbar to either of the areas in the payoff table displaying the strategy labels.



For example, dragging the player icon from the left of Bob's name in the list of players and dropping it on the right side of Alice's strategy label column changes the display of the game as in Here, the strategies are shown in a hierarchical format, enumerating the outcomes of the game first by Alice's (red) strategy choice, then by Bob's (blue) strategy choice.

Alternatively, the game can be displayed by listing the outcomes with Bob's strategy choice first, then Alice's. Drag Bob's player icon and drop it on the left side of Alice's strategy choices, and the game display changes to organize the outcomes first by Bob's action, then by Alice's.

The same dragging operation can be used to assign players to the columns. Assigning multiple players to the columns gives the same hierarchical presentation of those players' strategies. Dropping a player above another player's strategy labels assigns him to a higher level of the column player hierarchy; dropping a player below another player's strategy labels assigns him to a lower level of the column player hierarchy.



As the assignment of players in the row and column hierarchies changes, the ordering of the payoffs in each cell of the table also changes. In all cases, the color-coding of the entries identifies the player to whom each payoff corresponds. The ordering convention is chosen so that for a two player game in which one player is a row player and the other a column player, the row player's payoff is shown first, followed by the column player, which is the most common convention in print.

5.3.2 Adding players and strategies

To add an additional player to the game, use the menu item *Edit* → *Add player*, or the corresponding toolbar icon . The newly created player has one strategy, by default labeled with the number *1*.

Gambit supports arbitrary numbers of strategies for each player. To add a new strategy for a player, click the new strategy icon located to the left of that player's name.

To edit the names of strategies, click on any cell in the strategic game table where the strategy label appears, and edit the label using the edit control.

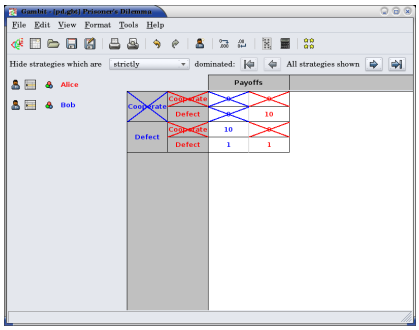
5.3.3 Editing payoffs

Payoffs for each player are specified individually for each contingency, or collection of strategies, in the game. To edit any payoff in the table, click that cell in the table and edit the payoff. Pressing the Escape key (`Esc`) cancels any editing of the payoff and restores the previous value.

To speed entry of many payoffs, as is typical when creating a new game, accepting a payoff entry via the `Tab` key automatically moves the edit control to the next cell to the right. If the payoff is the last payoff listed in a row of the table, the edit control wraps around to the first payoff in the next row; if the payoff is in the last row, the edit control wraps around to the first payoff in the first row. So a strategic game payoff table can be quickly entered by clicking on the first payoff in the upper-left cell of the table, inputting the payoff for the first (row) player, pressing the `Tab` key, inputting the payoff for the second (column) player, pressing the `Tab` key, and so forth, until all the payoff entries in the table have been filled.

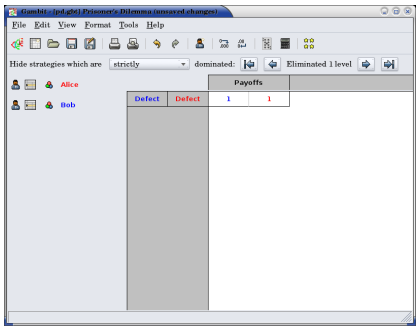
5.4 Investigating dominated strategies

Selecting *Tools* → *Dominance* toggles the appearance of a toolbar which can be used to investigate the structure of dominated strategies. Strategies can be eliminated iteratively based on whether they are strictly or weakly dominated.



When the dominance toolbar is shown, the strategic game table contains indicators of strategies that are dominated. In the prisoner's dilemma, the Cooperate strategy is strictly dominated for both players. This strict dominance is indicated by the solid "X" drawn across the corresponding strategy labels for both players. In addition, the payoffs corresponding to the dominated strategies are also drawn with a solid "X" across them. Thus, any contingency in the table containing at least one "X" is a contingency that can only be reached by at least one player playing a strategy that is dominated.

Strategies that are weakly dominated are similarly indicated, except the "X" shape is drawn using a thinner, dashed line instead of the thick, solid line.



Clicking the next level icon removes the strictly dominated strategies from the display.

The elimination of multiple levels can be automated using the fast forward icon, which iteratively eliminates dominated actions until no further actions can be eliminated. The rewind icon restores the display to the full game.

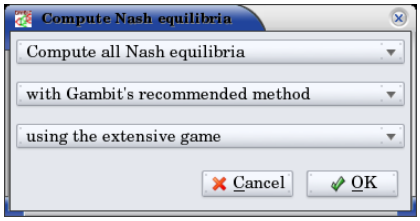
5.5 Computing Nash equilibria

Gambit offers broad support for computing Nash equilibria in both extensive and strategic games. To access the provided algorithms for computing equilibria, select *Tools* → *Equilibrium*, or click on the calculate icon on the toolbar.

5.5.1 Selecting the method of computing equilibria

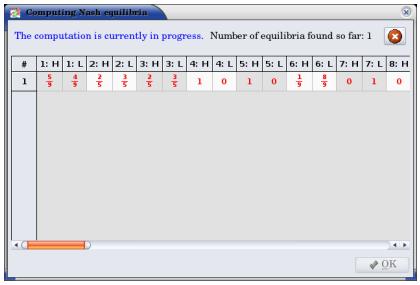
The process of computing Nash equilibria in extensive and strategic games is similar. This section focuses on the case of extensive games; the process for strategic games is analogous, except the extensive game-specific features, such as displaying the profiles on the game tree, are not applicable.

Gambit provides guidance on the options for computing Nash equilibria in a dialog. The methods applicable to a particular game depend on three criteria: the number of equilibria to compute, whether the computation is to be done on the extensive or strategic games, and on details of the game, such as whether the game has two players or more, and whether the game is constant-sum.



The first step in finding equilibria is to specify how many equilibria are to be found. Some algorithms for computing equilibria are adapted to finding a single equilibrium, while others attempt to compute the whole equilibrium set. The first drop-down in the dialog specifies how many equilibria to compute. In this drop-down there are options for *as many equilibria as possible* and, for two-player games, *all equilibria*. For some games, there exist algorithms which will compute many equilibria (relatively) efficiently, but are not guaranteed to find all equilibria.

To simplify this process of choosing the method to compute equilibria in the second drop-down, Gambit provides for any game "recommended" methods for computing one, some, and all Nash equilibria, respectively. These methods are selected based on experience as to the efficiency and reliability of the methods, and should generally work well on most games. For more control over the process, the user can select from the second drop-down in the dialog one of the appropriate methods for computing equilibria. This list only shows the methods which are appropriate for the game, given the selection of how many equilibria to compute. More details on these methods are contained in *CLI*.

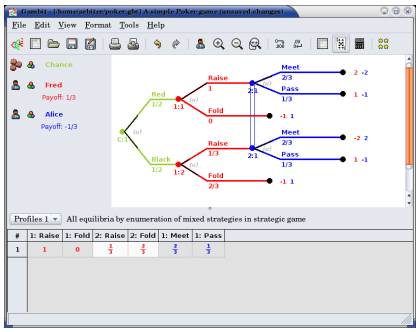


Finally, for extensive games, there is an option of whether to use the extensive or strategic game for computation. In general, computation using the extensive game is preferred, since it is often a significantly more compact representation of the strategic characeteristics of the game than the reduced strategic game is.

For even moderate sized games, computation of equilibrium can be a time-intensive process. Gambit runs all computations in the background, and displays a dialog showing all equilibria computed so far. The computation can be cancelled at any time by clicking on the cancel icon , which terminates the computation but keeps any equilibria computed.

5.5.2 Viewing computed profiles in the game

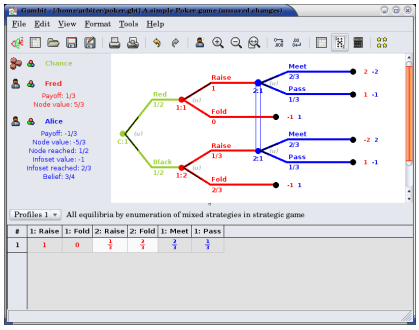
After computing equilibria, a panel showing the list of equilibria computed is displayed automatically. The display of this panel can be toggled by selecting *View* → *Profiles*, or clicking on the playing card icon on the toolbar.



This game has a unique equilibrium in which Fred raises after Red with probability one, and raises with probability one-third after Black. Alice, at her only information set, plays meet with probability two- thirds and raise with probability one-third.

This equilibrium is displayed in a table in the profiles panel. If more than one equilibrium is found, this panel lists all equilibria found. Equilibria computed are grouped by separate computational runs; computing equilibria using a different method (or different settings) will add a second list of profiles. The list of profiles displayed is selected using the drop-down at the top left of the profiles panel; in the screenshot, it is set to *Profiles 1*. A brief description of the method used to compute the equilibria is listed across the top of the profiles panel.

The currently selected equilibrium is shown in bold in the profiles listing, and information about this equilibrium is displayed in the extensive game. In the figure, the probabilities of selecting each action are displayed below each branch of the tree. (This is the default Gambit setting; see *Controlling the layout of the tree* for configuring the labeling of trees.) Each branch of the tree also shows a black line, the length of which is proportional to the probability with which the action is played.

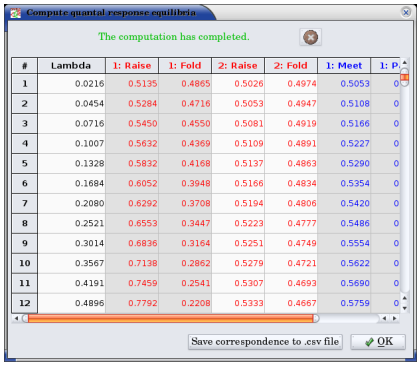


Clicking on any node in the tree displays additional information about the profile at that node. The player panel displays information relevant to the selected node, including the payoff to all players conditional on reaching the node, as well as information about Alice's beliefs at the node.

The computed profiles can also be viewed in the reduced strategic game. Clicking on the strategic game icon changes the view to the reduced strategic form of the game, and shows the equilibrium profiles converted to mixed strategies in the strategic game.

5.5.3 Computing quantal response equilibria

Gambit provides methods for computing the logit quantal response equilibrium correspondence for extensive games [McKPal98] and strategic games [McKPal95], using the tracing method of [Tur05].

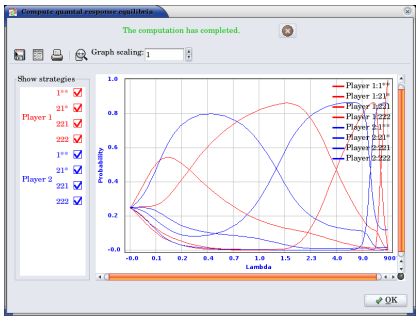


To compute the correspondence, select *Tools* → *Qre*. If viewing an extensive game, the agent quantal response equilibrium correspondence is computed; if viewing a strategic game (including the reduced strategic game derived from an extensive game), the correspondence is computed in mixed strategies.

The computed correspondence values can be saved to a CSV (comma- separated values) file by clicking the button labeled *Save correspondence to .csv file*. This format is suitable for reading by a spreadsheet or graphing application.

5.5.4 Quantal response equilibria in strategic games (experimental)

There is an experimental graphing interface for quantal response equilibria in strategic games. The graph by default plots the probabilities of all strategies, color- coded by player, as a function of the lambda parameter. The lambda values on the horizontal axis are plotted using a sigmoid transformation; the Graph scaling value controls the shape of this transformation. Lower values of the scaling give more graph space to lower values of lambda; higher values of the scaling give more space to higher values of lambda.



The strategies graphed are indicated in the panel at the left of the window. Clicking on the checkbox next to a strategy toggles whether it is displayed in the graph.

The data points computed in the correspondence can be viewed (as in the extensive game example above) by clicking on the show data icon on the toolbar. The data points can be saved to a CSV file by clicking on the .

To zoom in on a portion of the graph of interest, hold down the left mouse button and drag a rectangle on the graph. The plot window zooms in on the portion of the graph selected by that rectangle. To restore the graph view to the full graph, click on the zoom to fit icon .

To print the graph as shown, click on the print icon . Note that this is very experimental, and the output may not be very satisfactory yet.

5.6 Printing and exporting games

Gambit supports (almost) WYSIWYG (what you see is what you get) output of both extensive and strategic games, both to a printer and to several graphical formats. For all of these operations, the game is drawn exactly as currently displayed on the screen, including whether the extensive or strategic representation is used, the layout, colors for players, dominance and probability indicators, and so forth.

5.6.1 Printing a game

To print the game, press `Ctrl-P`, select *File* → *Print*, or click the printer icon on the toolbar. The game is scaled so that the printout fits on one page, while maintaining the same ratio of horizontal to vertical size; that is, the scaling factor is the same in both horizontal and vertical dimensions.

Note that especially for extensive games, one dimension of the tree is much larger than the other. Typically, the extent of the tree vertically is much greater than its horizontal extent. Because the printout is scaled to fit on one page, printing such a tree will generally result in what appears to be a thin line running vertically down the center of the page. This is in fact the tree, shrunk so the large vertical dimension fits on the page, meaning that the horizontal dimension, scaled at the same ratio, becomes very tiny.

5.6.2 Saving to a graphics file

Gambit supports export to five graphical file formats:

- Windows bitmaps (.bmp)
- JPEG, a lossy compressed format (.jpg , .jpeg)
- PNG, a lossless compressed format (.png); these are similar to GIFs
- Encapsulated PostScript (.ps)
- Scalable vector graphics (.svg)

To export a game to one of these formats, select *File* → *Export*, and select the corresponding menu entry.

The Windows bitmap and PNG formats are generally recommended for export, as they both are lossless formats, which will reproduce the game image exactly as in Gambit. PNG files use a lossless compression algorithm, so they are typically much smaller than the Windows bitmap for the same game. Not all image viewing and manipulation tools handle PNG files; in those cases, use the Windows bitmap output instead. JPEG files use a compression algorithm that only approximates the original version, which often makes it ill-suited for use in saving game images, since it often leads to "blocking" in the image file.

For all three of these formats, the dimensions of the exported graphic are determined by the dimensions of the game as drawn on screen. Image export is only supported for games which are less than about 65000 pixels in either the horizontal or vertical dimensions. This is unlikely to be a practical problem, since such games are so large they usually cannot be drawn in such a way that a human can make sense of them.

Encapsulated PostScript output is generally useful for inclusion in LaTeX and other scientific document preparation systems. This is a vector-based output, and thus can be rescaled much more effectively than the other output formats.

CHAPTER

SIX

CATALOG OF GAMES

Below is a complete list of games included in Gambit's catalog. Check out the *pygambit API reference* for instructions on how to search and load these games in Python, and the *Updating the games catalog* guide for instructions on how to contribute new games to the catalog.

Extensive form games

Bagwell (GEB 1995) commitment and (un)observability

This is a Stackelberg-type game with imperfectly observed commitment, following the analysis of Bag1995. The outcomes and payoffs are the same as in Bagwell's model. This example sets the probability that the follower 'correctly' observes the leader's action as .99 (99/100). The key result is that the only pure-strategy equilibrium that survives if observability is imperfect is the one in which players choose the actions that would form an equilibrium if the game was a *simultaneous-move* game. There is an equilibrium in which the 'Stackelberg' action is played with high probability, but strictly less than one.

Load in PyGambit:

```
pygambit.catalog.load("bagwell1995")
```

Download:
`bagwell1995.efg`

A simple Poker game

This is a simple game of one-card poker from Mye91, used as the introductory example for game models.
Note that as specified in the text, the game has the slightly unusual feature that folding with the high (red) card results in the player winning rather than losing.

See also Rei2008
Another one-card poker game where folding with the high card is a loss rather than a win.

Load in PyGambit:

```
pygambit.catalog.load("myerson1991/fig2_1")
```

Download:
`myerson1991/fig2_1.efg`

Myerson (1991) Figure 4.2

An example from Mye91 which illustrates the distinction between an equilibrium of an extensive form game and an equilibrium of its (multi)agent representation. The actions B1, Z1, and W2 form a behavior profile which is an equilibrium in the (multi)agent representation. However, it is not a Nash equilibrium of the extensive game, because Player 1 would prefer to switch from (B1, Z1) to (A1, Y1); the (multi)agent representation rules out such coordinated deviations across information sets.

Load in PyGambit:

```
pygambit.catalog.load("myerson1991/fig4_2")
```

Download:
`myerson1991/fig4_2.efg`

Stripped-down poker (Reiley et al 2008)

This is a one-card poker game used in Rei2008 as a teaching exercise.

See also Mye91
Another one-card poker game with slightly different rules.

Load in PyGambit:

```
pygambit.catalog.load("reiley2008/fig1")
```

Download:
`reiley2008/fig1.efg`

Selten's horse (Selten IJGT 1975, Figure 1)

This is a three-player game presented in Sel75, commonly referred to as "Selten's horse" owing to the layout in which it can be drawn. It is the motivating example for his definition of (trembling-hand) perfect equilibrium, by showing a game that has an equilibrium which is "unreasonable", but which is not ruled out by subgame perfection because this game has no proper subgames.

Load in PyGambit:

```
pygambit.catalog.load("selten1975/fig1")
```

Download:
`selten1975/fig1.efg`

CHAPTER

SEVEN

DEVELOPER DOCS

This section contains information for developers who want to contribute to the Gambit project, including how to build Gambit from source, how to contribute code, and how to report bugs.

7.1 Building Gambit from source

This page covers instructions for building Gambit from source. This is for those who are interested in developing Gambit, or who want to play around with the latest features before they make it into a pre-compiled binary version.

This page requires at least some familiarity with programming. Most users will want to stick with released distributions; see the *Install* page for how to get the current version for your operating system. Following the instructions here will install the Gambit CLI, GUI and Python extension (PyGambit).

The steps you will need to follow to build from source are as follows:

1. Refer to the *contributor page* which explains how to clone the Gambit repository from GitHub (you may first wish to create a fork).
2. *Install the necessary build tools and dependencies for your platform*.
3. *Follow the platform-specific instructions to build and install Gambit CLI and GUI components from source*.
4. *Build the Python extension (PyGambit)*.

7.1.1 Install build tools and dependencies

Install on macOS via Homebrew

1. Check that you have Homebrew installed by running *brew –version*. If not, follow the instructions at https://brew.sh/.
2. Install build dependencies:

```
brew install automake autoconf libtool wxwidgets
```

> **Note**
>
> If you encounter interpreter errors with autom4te, you may need to ensure your Perl installation is correct or reinstall the autotools:
>
> ```
> brew reinstall automake autoconf libtool wxwidgets
> ```

Install on Linux (Debian/Ubuntu) via apt

1. Update your package lists:

```
sudo apt update
```

2. Install general build dependencies:

```
sudo apt install build-essential automake autoconf libtool
```

3. Install GUI dependencies:

```
sudo apt install libwxgtk3.2-dev
```

Install on Windows

The recommended way to build Gambit on modern Windows is to use the MSYS2 / MinGW-w64 environment.

1. Download and install MSYS2 from https://www.msys2.org/ and follow the update instructions there (you will typically run `pacman -Syu` and restart the MSYS2 terminal as instructed).
2. Open the "MSYS2 MinGW 64-bit" terminal (important: use the MinGW shell, not the plain MSYS shell).
3. Install general build dependencies

```
# update package DB & core packages first (may require closing/reopening the shell)
pacman -Syu

# install compiler toolchain, autotools and libtool
pacman -S --needed mingw-w64-x86_64-toolchain mingw-w64-x86_64-automake \
mingw-w64-x86_64-autoconf mingw-w64-x86_64-libtool mingw-w64-x86_64-make
```

4. Install GUI dependencies

```
pacman -S --needed mingw-w64-x86_64-wxwidgets3.2
```

> **Note**
>
> When building for a different target (32-bit) substitute the corresponding MinGW packages (`mingw-w64-i686-*`).

7.1.2 Install CLI and GUI from source

Navigate to the Gambit source directory (use the "MSYS2 MinGW 64-bit" terminal on Windows) and run:

```
aclocal
libtoolize
automake --add-missing
autoconf
./configure
make
# Skip this on Windows:
sudo make install
```

Build macOS application bundle

1. Create macOS application bundle:

   To create a distributable DMG file:

   ```
   make osx-dmg
   ```

2. Install the application:

   After creating the DMG file, open it and drag the Gambit application to your Applications folder.

Creating a Windows installer

1. Create a `.msi` installer. This requires the Wix toolset.

   ```
   make msw-msi
   ```

2. Install the application:

   Run the generated `gambit-X.Y.Z.msi` file to install Gambit on your system.

Note

Command-line options are available to modify the configuration process; do *./configure –help* for information. Of these, the option which may be most useful is to disable the build of the graphical interface.

By default Gambit will be installed in /usr/local. You can change this by replacing configure step with one of the form

```
./configure --prefix=/your/path/here
```

Note

If you don't want to build the graphical interface, you can pass the argument `--disable-gui` to the configure step, for example:

```
./configure --disable-gui
```

Warning

If wxWidgets isn't installed in a standard place (e.g., `/usr` or `/usr/local`), you'll need to tell `configure` where to find it with the `--with-wx-prefix=PREFIX` option, for example:

```
./configure --with-wx-prefix=/home/mylogin/wx
```

Warning

The graphical interface relies on external calls to other programs built in this process, especially for the computation of equilibria. It is strongly recommended that you install the Gambit executables to a directory in your path!

7.1.3 Building the Python extension

The *pygambit Python package* is in `src/pygambit` in the Gambit source tree. We recommend to install *pygambit* as part of a virtual environment rather than in the system's Python (for example using *venv*). Use *pip* to install from the **root directory of the source tree**:

```
python -m venv venv
source venv/bin/activate
python -m pip install ".[test,doc]"
```

> **Tip**
>
> The "test" and "doc" optional dependencies are useful for developers wishing to run the test suite or build this documentation locally.

Once installed, simply `import pygambit` in your Python shell or script to get started.

7.2 Contributing to Gambit

This section provides guidelines for contributing to Gambit, including how to report bugs, suggest features, and contribute code. It includes information relevant to both core developers and external contributors.

7.2.1 Code of Conduct

All participants in the Gambit community are expected to show respect and courtesy to others. We are committed to fostering a welcoming and harassment-free environment for everyone, regardless of background or identity. Please be kind and considerate in your interactions. We are all here to build a better tool for game theory research and education. Disagreements may happen, but they should be handled with respect and a focus on constructive resolution.

To raise any concerns, please contact *Ted Turocy <mailto:ted.turocy@gmail.com>* or *Rahul Savani <mailto:rahul.savani@gmail.com>*.

7.2.2 GitHub issues

Newcomer to the project? Please read the *code of conduct* and *new contributors section* sections before posting on GitHub.

In the first instance, bug reports, feature requests and improvements to the Gambit documentation should be posted to the Gambit issue tracker, located at http://github.com/gambitproject/gambit/issues. Use the issue templates to help you provide the necessary information.

When reporting a bug, please be sure to include the following:

- The version(s) of Gambit you are using. (If possible, it is helpful to know whether a bug exists in both the current stable/teaching and the current development/research versions.)
- The operating system(s) on which you encountered the bug.
- A detailed list of steps to reproduce the bug. Be sure to include a sample game file or files if appropriate; it is often helpful to simplify the game if possible.

7.2.3 Contributing code

Gambit is an open-source project, and contributions are welcome from anyone. The project is hosted on GitHub, and contributions can be made via pull requests following the standard GitHub workflow.

In the git repository, the branch `master` always points to the latest development version. New development should in general always be based off this branch. Branches labeled `maintX_Y`, where `X` is the major version number and `Y` is the minor version number, point to the latest commit on a stable version.

A Note for New Contributors

We warmly welcome new contributors to the Gambit project! Your help is valuable to us.

Before you start working on a contribution, we encourage you to familiarize yourself with the project. Gambit is a mature and complex codebase. Even issues marked as "good first issue" may require some understanding of the context and how different parts of the project interact.

We have always valued thoughtful contributions. A common issue for open-source projects is receiving pull requests that are not well-aligned with the project's needs or coding standards. To avoid unproductive effort on your part and on ours, we strongly encourage you to engage with us before you spend a lot of time on implementation.

A great way to start is by:

1. Looking through the code to understand the relevant parts for your intended change.
2. Contacting the maintainers via commenting with a question on an open issue, or opening a Discussion on GitHub.
3. After your discussion with the maintainers, open a draft pull request early in your process. This allows for discussion and feedback before you've invested too much time.

We are happy to answer questions and provide guidance. Engaging with the maintainers early ensures that your contribution is likely to be accepted and integrated smoothly.

Policy on AI-Assisted Contributions

We recognize that generative AI tools can be a useful aid in software development. We also expect authentic and thoughtful engagement from our contributors.

- **You are responsible for your contributions.** If you use generative AI to help you write code or documentation, you must fully understand the output. You should be able to explain the changes and why they are the correct approach for the project.
- **Add value.** Simply taking a prompt, feeding it to an AI, and posting the result as a contribution is not helpful. We expect you to use your own expertise to verify, test, and refine any AI-generated content. Out of respect for everyone's time, we reserve the right to rigorously reject low-value contributions, whether AI-generated or not.
- **No AI-generated comments.** Please do not post output from Large Language Models (LLMs) or similar tools as comments on GitHub issues or pull requests. Such comments are often generic and do not add to the discussion.
- **Humans over bots.** We discourage the use of automated tools, such as bots or agents, to post AI-generated content to issues or pull requests, without the advance approval of the core development team.

How to submit a contribution

To contribute code, please follow these steps:

1. To get started contributing code in the Gambit GitHub repo, do one of the following:
   - Core developers: request contributor access from one of the team
   - External contributors: fork the repository on GitHub.
2. Clone the repository to your local machine

```
git clone https://github.com/gambitproject/gambit.git  # or your fork URL
cd gambit
```

3. Follow the instructions in the *Building Gambit from source* page to set up your development environment and build Gambit from source. If you only plan to make changes to the PyGambit Python code, you can skip to *Building the Python extension*.

4. *[Optional but recommended]* Install *pre-commit* which is used to run code formatters and linters before each commit. This helps ensure code quality and consistency. You can install it into the virtual environment where you installed PyGambit

   ```
   pip install pre-commit
   ```

   Alternatively, *pre-commit* is also available via various packaging systems. For example, you can install via Homebrew

   ```
   brew install pre-commit
   ```

   If you install via a package manager (instead of into the virtual environment), *pre-commit* will be available for use with other projects.

   Then, set it up in the Gambit repository

   ```
   pre-commit install
   ```

   Having *pre-commit* installed is recommended as it runs many of the same checks that are automatically conducted on any pull request. Every time you commit, it will automatically fix some issues and highlight others for manual adjustment.

5. Create a new branch for your changes. It's good practice to either give the branch a descriptive name or directly reference an issue number

   ```
   git checkout -b feature/issue-number
   ```

6. Make your changes.

   ```
   git add <files>
   git commit -m "Clear and descriptive commit message"
   ```

   Provide a clear commit message. Gambit does not have its own set of guidelines for commit messages. However, there are a number of webpages that have suggestions for writing effective commit messages (and for deciding how to structure your contributions as one or more commits as appropriate). See for example this page.

7. Push your changes to your fork or branch

   ```
   git push origin feature/issue-number
   ```

8. Open a pull request on GitHub to the master branch of the upstream repository. Ensure your pull request:

   - Has an informative title.
   - Links to any relevant issues.
   - Includes a clear description of the changes made.
   - Explains how a reviewer can test the changes.

> **Note**
>
> It's good practice to open a draft pull request early in the development process to facilitate discussion and feedback, ensure there are no merge conflicts, and allow for continuous integration testing via GitHub Actions.

9. Core developers will review your changes, provide feedback, and merge them into the master branch if they meet the project's standards.
10. Once your pull request is merged, delete your branch on GitHub (a button should appear to do this automatically).

7.2.4 Testing your changes

Be sure to familiarise yourself with *Contributing code* before reading this section.

By default, pull requests on GitHub will trigger the running of Gambit's test suite using GitHub Actions. You can also run the tests locally before submitting your pull request, using *pytest*.

1. Ensure *pygambit* is installed with test dependencies: see *Building the Python extension*.
2. Run pytest:

```
pytest --run-tutorials
```

> **Tip**
>
> You can omit the *–run-tutorials* to skip running the tutorial notebook tests which take longest to run. Running tests including tutorials requires *doc* as well as *test* dependencies; see *Building the Python extension*.

Adding to the test suite

Tests can be added to the test suite by creating new test files in the `tests` directory. Tests should be written using the *pytest* framework. Refer to existing test files for examples of how to write tests or see the pytest documentation for more information.

7.2.5 Editing this documentation

Be sure to familiarise yourself with *Contributing code* before reading this section.

You can make changes to the documentation by editing the *.rst* files in the `doc` directory. Creating a pull request with your changes will automatically trigger a build of the documentation via the ReadTheDocs service, which can be viewed online. You can also build the documentation locally to preview your changes before submitting a pull request.

1. Install Pandoc for your OS
2. Ensure *pygambit* is installed with doc dependencies: see *Building the Python extension*.
3. Navigate to the Gambit repo and build the docs:

```
cd doc
make html  # or make livehtml for live server with auto-rebuild
```

4. Open `doc/_build/html/index.html` in your browser to view the documentation.

Contributing tutorials

To submit a tutorial for inclusion in the Gambit documentation, please follow these steps:

1. Open a GitHub issue using the *Tutorial request* issue template on the Gambit GitHub repo or choose an issue already opened with the *tutorial* label.
2. Write the tutorial as a Jupyter notebook (*.ipynb* file), following the style and format of existing tutorials in the *doc/tutorials* directory. Develop this on a branch as per the instructions in *Contributing code*. Add the tutorial to the *doc/tutorials* directory in the repository. Put it in an appropriate subdirectory or create a new one if necessary.
3. Update *doc/pygambit.rst* to ensure the tutorial is listed in the docs at an appropriate location.
4. *[Optional]* If your tutorial requires additional dependencies not already listed in the `doc` list under `[project.optional-dependencies]` inside `pyproject.toml`, please add them to the file.

7.2.6 Recognising contributions

Gambit is set up with All Contributors to recognise all types of contributions, including code, documentation, bug reports, and more.

You can see the list of contributors on the README page of the Gambit GitHub repo.

To add a contributor, comment on a GitHub Issue or Pull Request, asking @all-contributors to add a contributor:

> @all-contributors please add @<username> for <contributions>

Refer to the emoji key for a list of contribution types.

7.2.7 Releases & maintenance branches

Releases of Gambit are made by core developers. Details of previous releases can be found in the GitHub releases page.

Branches labeled `maintX` and `maintX_Y`, where `X` is the major version number and `Y` is the minor version number, are associated with releases and point to the latest commit on a stable version. Navigate to the Gambit repository on GitHub and select the *branches* tab to see the list of active maintenance branches. Be sure to delete any maintenance branches that are no longer being maintained.

Making a new release

When making a new release of Gambit, follow these steps:

1. Create a new branch from the latest commit on the `master` branch named `maintX_Y`, where `X` is the major version number and `Y` is the minor version number of the new release.
2. Update the version number in the `build_support/GAMBIT_VERSION` file to `X.Y.Z`.

   All other files will automatically use the updated version number:

   - *pyproject.toml* reads from GAMBIT_VERSION file at build time
   - *configure.ac* reads from GAMBIT_VERSION file and substitutes into *build_support/osx/Info.plist* and *build_support/msw/gambit.wxs*
   - *src/pygambit/__init__.py* reads from installed package metadata or GAMBIT_VERSION file
   - *doc/conf.py* reads from GAMBIT_VERSION file at documentation build time
   - Documentation pages reference the |*release*| substitution variable to automatically reflect the updated version number.
3. Update the *ChangeLog* file with a summary of changes

4. Once there are no further commits to be made for the release, create a tag for the release from the latest commit on the maintenance branch.

```
git tag -a vX.Y.Z -m "Gambit version X.Y.Z"
```

5. Push the maintenance branch and tags to the GitHub repository.

```
git push origin maintX_Y
git push origin --tags
```

6. Create a new release on the GitHub releases page, using the tag created in step 4. Include a summary of changes from the *ChangeLog* file in the release notes.
7. Currently there is no automated process for pushing the new release to PyPI. This must be done manually.

Patching maintained versions

If you have bug fixes that should be applied across maintained versions:

1. Make a branch from the oldest maintenance branch to which the change applies
2. Apply the change and make a pull request to that maintenance branch
3. After the pull request is merged, open new pull requests to each subsequent maintenance branch and finally to the master branch, merging each in turn
4. Create new releases from each maintenance branch as needed, following the steps in *Making a new release*.

7.3 Updating the Games Catalog

This page includes developer notes regarding the catalog module, and the process for contributing to and updating Gambit's *Games Catalog*. To do so, you will need to have the *gambit* GitHub repo cloned and be able to submit pull request via GitHub; you may wish to first review the *contributor guidelines*. You'll also need to have a developer install of *pygambit* available in your Python environment, see *Building the Python extension*.

7.3.1 Add new game files

You can add games to the catalog saved in a valid representation *format*. Currently supported representations are:

- *.efg* for extensive form games
- *.nfg* for normal form games

1. **Create the game file:**

   Use either *pygambit*, the Gambit *CLI* or *GUI* to create and save game in a valid representation *format*. Make sure the game includes a description, with any citations referencing the *bibliography*. Use a full link to the bibliography entry, so the link can be accessed from the file directly, as well as being rendered in the docs e.g. `` `Rei2008 <https://gambitproject.readthedocs.io/en/latest/biblio.html#Rei2008>`_ ``

2. **Add the game file:**

   Create a new branch in the `gambit` repo. Add your new game file(s) inside the `catalog` dir and commit them.

3. **Update the catalog:**

   Reinstall the package to pick up the new game file(s) in the `pygambit.catalog` module. Then use the `update.py` script to update Gambit's documentation & build files.

```
pip install .
python build_support/catalog/update.py --build
```

> **Warning**
>
> Running the script with the `--build` flag updates *Makefile.am*. If you moved games that were previously in *contrib/games* you'll need to also manually remove those files from *EXTRA_DIST*.

4. **Submit a pull request to GitHub with all changes.**

> **Warning**
>
> Make sure you commit all changed files e.g. run `git add --all` before committing and pushing.

7.3.2 Access from pygambit

We keep the `catalog` directory at the top level of the repository because it is in principle independent of the Python and C++ code. However, in order to include these games with the Python package, there is a bit of extra infrastructure.

In `setup.py` we have a custom build command which first copies the contents of `catalog/` into the build directory for the Python package. These are then exposed as data in the `catalog_data` directory (changing the name to avoid confusion or clashes with `catalog.py`, which is responsible for accessing the catalog).

The main implication is that if you are working via the Python package and you add new games to the catalog, you will need to rebuild and reinstall the Python extension in order to access the new games. That is, changing the contents of the catalog is no different than changing any other source code in the Python package; you'll need to execute `pip install .` after the addition or change.

CHAPTER

EIGHT

GAME REPRESENTATION FORMATS

This section documents the file formats recognized by Gambit. These file formats are text-based and designed to be readable and editable by hand by humans to the extent possible, although programmatic tools to generate and manipulate these files are almost certainly needed for all but the most trivial of games.

These formats can be viewed as being low-level. They define games explicitly in terms of their structure, and do not support any sort of parameterization, macros, and the like. Thus, they are adapted largely to the type of input required by the numerical methods for computing Nash equilibria, which only apply to a particular realization of a game's parameters. Higher-level tools, whether the graphical interface or scripting applications, are indicated for doing parametric analysis and the like.

8.1 Conventions common to all file formats

Several conventions are common to the interpretation of the file formats listed below.

Whitespace is not significant. In general, whitespace (carriage returns, horizontal and vertical tabs, and spaces) do not have an effect on the meaning of the file. The only exception is inside explicit double-quotes, where all characters are significant. The formatting shown here is the same as generated by the Gambit code and has been chosen for its readability; other formattings are possible (and legal).

Text labels. Most objects in an extensive game may be given textual labels. These are prominently used in the graphical interface, for example, and it is encouraged for users to assign nonempty text labels to objects if the game is going to be viewed in the graphical interface. In all cases, these labels are surrounded by the quotation character (“). The use of an explicit “ character within a text label can be accomplished by preceding the embedded “ characters with a backwards slash (). This is an alternate version of the first line of the example file, in which the title of the game contains the term Bayesian game in quotation marks:

```
EFG 2 R "An example of a \"Bayesian game\"" { "Player 1" "Player 2" }
```

Numerical data. Numerical data, namely, the payoffs at outcomes, and the action probabilities for chance nodes, may be expressed in integer, decimal, or rational formats. In all cases, numbers are understood by Gambit to be exact, and represented as such internally. For example, the numerical entries 0.1 and 1/10 represent the same quantity.

In versions 0.97 and prior, Gambit distinguished between floating point and rational data. In these versions, the quantity 0.1 was represented interally as a floating-point number. In this case, since 0.1 does not have an exact representation in binary floating point, the values 0.1 and 1/10 were not identical, and some methods for computing equilibria could give (slightly) different results for games using one versus the other. In particular, using rational-precision methods on games with the floating point numbers could give unexpected output, since the conversion of 0.1 first to floating-point then to rational would involve roundoff error. This is largely of technical concern, and the current Gambit implementation now behaves in such a way as to give the “expected” result when decimal numbers appear in the file format.

8.1.1 The extensive game (.efg) file format

The extensive game (.efg) file format has been used by Gambit, with minor variations, to represent extensive games since circa 1994. It replaced an earlier format, which had no particular name but which had the conventional extension .dt1.

A sample file

This is a sample file illustrating the general format of the file:

```
EFG 2 R "General Bayes game, one stage" { "Player 1" "Player 2" }
c "ROOT" 1 "(0,1)" { "1G" 0.500000 "1B" 0.500000 } 0
c "" 2 "(0,2)" { "2g" 0.500000 "2b" 0.500000 } 0
p "" 1 1 "(1,1)" { "H" "L" } 0
p "" 2 1 "(2,1)" { "h" "l" } 0
t "" 1 "Outcome 1" { 10.000000 2.000000 }
t "" 2 "Outcome 2" { 0.000000 10.000000 }
p "" 2 1 "(2,1)" { "h" "l" } 0
t "" 3 "Outcome 3" { 2.000000 4.000000 }
t "" 4 "Outcome 4" { 4.000000 0.000000 }
p "" 1 1 "(1,1)" { "H" "L" } 0
p "" 2 2 "(2,2)" { "h" "l" } 0
t "" 5 "Outcome 5" { 10.000000 2.000000 }
t "" 6 "Outcome 6" { 0.000000 10.000000 }
p "" 2 2 "(2,2)" { "h" "l" } 0
t "" 7 "Outcome 7" { 2.000000 4.000000 }
t "" 8 "Outcome 8" { 4.000000 0.000000 }
c "" 3 "(0,3)" { "2g" 0.500000 "2b" 0.500000 } 0
p "" 1 2 "(1,2)" { "H" "L" } 0
p "" 2 1 "(2,1)" { "h" "l" } 0
t "" 9 "Outcome 9" { 4.000000 2.000000 }
t "" 10 "Outcome 10" { 2.000000 10.000000 }
p "" 2 1 "(2,1)" { "h" "l" } 0
t "" 11 "Outcome 11" { 0.000000 4.000000 }
t "" 12 "Outcome 12" { 10.000000 2.000000 }
p "" 1 2 "(1,2)" { "H" "L" } 0
p "" 2 2 "(2,2)" { "h" "l" } 0
t "" 13 "Outcome 13" { 4.000000 2.000000 }
t "" 14 "Outcome 14" { 2.000000 10.000000 }
p "" 2 2 "(2,2)" { "h" "l" } 0
t "" 15 "Outcome 15" { 0.000000 4.000000 }
t "" 16 "Outcome 16" { 10.000000 0.000000 }
```

Structure of the prologue

The extensive gamefile consists of two parts: the prologue, or header, and the list of nodes, or body. In the example file, the prologue is the first line. (Again, this is just a consequence of the formatting we have chosen and is not a requirement of the file structure itself.)

The prologue is constructed as follows. The file begins with the token EFG , identifying it as an extensive gamefile. Next is the digit 2 ; this digit is a version number. Since only version 2 files have been supported for more than a decade, all files have a 2 in this position. Next comes the letter R . The letter R used to distinguish files which had rational numbers for numerical data; this distinction is obsolete, so all new files should have R in this position.

The prologue continues with the title of the game. Following the title is a list of the names of the players defined in the game. This list follows the convention found elsewhere in the file of being surrounded by curly braces and delimited

by whitespace (but not commas, semicolons, or any other character). The order of the players is significant; the first entry in the list will be numbered as player 1, the second entry as player 2, and so forth. At the end of the prologue is an optional text comment field.

Structure of the body (list of nodes)

The body of the file lists the nodes which comprise the game tree. These nodes are listed in the prefix traversal of the tree. The prefix traversal for a subtree is defined as being the root node of the subtree, followed by the prefix traversal of the subtree rooted by each child, in order from first to last. Thus, for the whole tree, the root node appears first, followed by the prefix traversals of its child subtrees. For convenience, the game above follows the convention of one line per node.

Each node entry begins with an unquoted character indicating the type of the node. There are three node types:

- *c* for a chance node
- *p* for a personal player node
- *t* for a terminal node

Each node type will be discussed individually below. There are three numbering conventions which are used to identify the information structure of the tree. Wherever a player number is called for, the integer specified corresponds to the index of the player in the player list from the prologue. The first player in the list is numbered 1, the second 2, and so on. Information sets are identified by an arbitrary positive integer which is unique within the player. Gambit generates these numbers as 1, 2, etc. as they appear first in the file, but there are no requirements other than uniqueness. The same integer may be used to specify information sets for different players; this is not ambiguous since the player number appears as well. Finally, outcomes are also arbitrarily numbered in the file format in the same way in which information sets are, except for the special number 0 which is reserved to indicate the null outcome. Outcome 0 must not have a name or payoffs specified.

Information sets and outcomes may (and frequently will) appear multiple times within a game. By convention, the second and subsequent times an information set or outcome appears, the file may omit the descriptive information for that information set or outcome. Alternatively, the file may specify the descriptive information again; however, it must precisely match the original declaration of the information set or outcome. Any mismatch in repeated declarations is an error, and the file is not valid. If any part of the description is omitted, the whole description must be omitted.

Outcomes may appear at nonterminal nodes. In these cases, payoffs are interpreted as incremental payoffs; the payoff to a player for a given path through the tree is interpreted as the sum of the payoffs at the outcomes encountered on that path (including at the terminal node). This is ideal for the representation of games with well- defined"stages"; see, for example, the file bayes2a.efg in the Gambit distribution for a two-stage example of the Bayesian game represented previously.

In the following lists, fields which are omittable according to the above rules are indicated by the label (optional).

Format of chance (nature) nodes. Entries for chance nodes begin with the character c . Following this, in order, are

- a text string, giving the name of the node
- a positive integer specifying the information set number
- (optional) the name of the information set and a list of actions at the information set with their corresponding probabilities
- a nonnegative integer specifying the outcome
- (optional) the name of the outcome and the payoffs to each player for the outcome

Format of personal (player) nodes. Entries for personal player decision nodes begin with the character p . Following this, in order, are:

- a text string, giving the name of the node
- a positive integer specifying the player who owns the node

- a positive integer specifying the information set
- (optional) the name of the information set and a list of action names for the information set
- a nonnegative integer specifying the outcome
- (optional) the name of the outcome and the payoffs to each player for the outcome

Format of terminal nodes. Entries for terminal nodes begin with the character t . Following this, in order, are:

- a text string, giving the name of the node
- a nonnegative integer specifying the outcome
- (optional) the name of the outcome and the payoffs to each player for the outcome

There is no explicit end-of-file delimiter for the file.

8.1.2 The strategic game (.nfg) file format, payoff version

This file format defines a strategic N-player game. In this version, the payoffs are listed in a tabular format. See the next section for a version of this format in which outcomes can be used to identify an equivalence among multiple strategy profiles.

A sample file

This is a sample file illustrating the general format of the file. This file is distributed in the Gambit distribution under the name e02.nfg:

```
NFG 1 R "Selten (IJGT, 75), Figure 2, normal form"
{ "Player 1" "Player 2" } { 3 2 }

1 1 0 2 0 2 1 1 0 3 2 0
```

Structure of the prologue

The prologue is constructed as follows. The file begins with the token NFG , identifying it as a strategic gamefile. Next is the digit 1 ; this digit is a version number. Since only version 1 files have been supported for more than a decade, all files have a 1 in this position. Next comes the letter R . The letter R used to distinguish files which had rational numbers for numerical data; this distinction is obsolete, so all new files should have R in this position.

The prologue continues with the title of the game. Following the title is a list of the names of the players defined in the game. This list follows the convention found elsewhere in the file of being surrounded by curly braces and delimited by whitespace (but not commas, semicolons, or any other character). The order of the players is significant; the first entry in the list will be numbered as player 1, the second entry as player 2, and so forth.

Following the list of players is a list of positive integers. This list specifies the number of strategies available to each player, given in the same order as the players are listed in the list of players.

The prologue concludes with an optional text comment field.

Structure of the body (list of payoffs)

The body of the format lists the payoffs in the game. This is a "flat" list, not surrounded by braces or other punctuation.

The assignment of the numeric data in this list to the entries in the strategic game table proceeds as follows. The list begins with the strategy profile in which each player plays their first strategy. The payoffs to all players in this contingency are listed in the same order as the players are given in the prologue. This, in the example file, the first two payoff entries are 1 1 , which means, when both players play their first strategy, player 1 receives a payoff of 1, and player 2 receives a payoff of 1.

Next, the strategy of the first player is incremented. Thus, player 1's strategy is incremented to his second strategy. In this case, when player 1 plays his second strategy and player 2 his first strategy, the payoffs are 0 2 : a payoff of 0 to player 1 and a payoff of 2 to player 2.

Now the strategy of the first player is again incremented. Thus, the first player is playing his third strategy, and the second player his first strategy; the payoffs are again 0 2 .

Now, the strategy of the first player is incremented yet again. But, the first player was already playing strategy number 3 of 3. Thus, his strategy now "rolls over" to 1, and the strategy of the second player increments to 2. Then, the next entries 1 1 correspond to the payoffs of player 1 and player 2, respectively, in the case where player 1 plays his second strategy, and player 2 his first strategy.

In general, the ordering of contingencies is done in the same way that we count: incrementing the least-significant digit place in the number first, and then incrementing more significant digit places in the number as the lower ones "roll over." The only differences are that the counting starts with the digit 1, instead of 0, and that the "base" used for each digit is not 10, but instead is the number of strategies that player has in the game.

8.1.3 The strategic game (.nfg) file format, outcome version

This file format defines a strategic N-player game. In this version, the payoffs are defined by means of outcomes, which may appear more than one place in the game table. This may give a more compact means of representing a game where many different strategy combinations map to the same consequences for the players. For a version of this format in which payoffs are listed explicitly, without identification by outcomes, see the previous section.

A sample file

This is a sample file illustrating the general format of the file. This file defines the same game as the example in the previous section:

```
NFG 1 R "Selten (IJGT, 75), Figure 2, normal form" { "Player 1" "Player 2" }

{
{ "1" "2" "3" }
{ "1" "2" }
}

{
{ "" 1, 1 }
{ "" 0, 2 }
{ "" 0, 2 }
{ "" 1, 1 }
{ "" 0, 3 }
{ "" 2, 0 }
}
1 2 3 4 5 6
```

Structure of the prologue

The prologue is constructed as follows. The file begins with the token NFG , identifying it as a strategic gamefile. Next is the digit 1 ; this digit is a version number. Since only version 1 files have been supported for more than a decade, all files have a 1 in this position. Next comes the letter R . The letter R used to distinguish files which had rational numbers for numerical data; this distinction is obsolete, so all new files should have R in this position.

The prologue continues with the title of the game. Following the title is a list of the names of the players defined in the game. This list follows the convention found elsewhere in the file of being surrounded by curly braces and delimited by whitespace (but not commas, semicolons, or any other character). The order of the players is significant; the first entry in the list will be numbered as player 1, the second entry as player 2, and so forth.

Following the list of players is a list of strategies. This is a nested list; each player's strategies are given as a list of text labels, surrounded by curly braces.

The nested strategy list is followed by an optional text comment field.

The prologue closes with a list of outcomes. This is also a nested list. Each outcome is specified by a text string, followed by a list of numerical payoffs, one for each player defined. The payoffs may optionally be separated by commas, as in the example file. The outcomes are implicitly numbered in the order they appear; the first outcome is given the number 1, the second 2, and so forth.

Structure of the body (list of outcomes)

The body of the file is a list of outcome indices. These are presented in the same lexicographic order as the payoffs in the payoff file format; please see the documentation of that format for the description of the ordering. For each entry in the table, a nonnegative integer is given, corresponding to the outcome number assigned as described in the prologue section. The special outcome number 0 is reserved for the "null" outcome, which is defined as a payoff of zero to all players. The number of entries in this list, then, should be the same as the product of the number of strategies for all players in the game.

8.1.4 The action graph game (.agg) file format

Action graph games (AGGs) are a compact representation of simultaneous-move games with structured utility functions. For more information on AGGs, the following paper gives a comprehensive discussion.

> A.X. Jiang, K. Leyton-Brown and N. Bhat, Action-Graph Games, Games and Economic Behavior, Volume 71, Issue 1, January 2011, Pages 141-173.

Each file in this format describes an action graph game. In order for the file to be recognized as AGG by GAMBIT, the initial line of the file should be:

```
#AGG
```

The rest of the file consists of 8 sections, separated by whitespaces. Lines with starting '#' are treated as comments and are allowed between sections.

1. The number of players, n.
2. The number of action nodes, |S|.
3. The number of function nodes, |P|.
4. Size of action set for each player. This is a row of n integers:

   $|S_1|$ $|S_2|$ .... $|S_n|$
5. Each Player's action set. We have N rows; row i has $|S_i|$ integers in ascending order, which are indices of Action nodes. Action nodes are indexed from 0 to |S|-1.
6. The Action Graph. We have |S| + |P| nodes, indexed from 0 to |S| + |P|-1. The function nodes are indexed after the action nodes. The graph is represented as (|S| + |P|) neighbor lists, one list per row. Rows 0 to |S| - 1 are for action nodes; rows |S| to |S| + |P|-1 are for function nodes. In each row, the first number |v| specifies the number of neighbors of the node. Then follows |v| numbers, corresponding to the indices of the neighbors.

   We require that each function node has at least one neighbor, and the neighbors of function nodes are action nodes. The action graph restricted to the function nodes has to be a directed acyclic graph (DAG).
7. Signatures of functions. This is |P| rows, each specifying the mapping f_p that maps from the configuration of the function node p's neighbors to an integer for p's "action count". Each function is specified by its "signature" consisting of an integer type, possibly followed by further parameters. Several types of mapping are implemented:

   - Types 0-3 require no further input.

  - Type 0: Sum. i.e. The action count of a function node p is the sum of the action counts of p's neighbors.
  - Type 1: Existence: boolean for whether the sum of the counts of neighbors are positive.
  - Type 2: The index of the neighbor with the highest index that has non-zero counts, or |S| + |P| if none applies.
  - Type 3: The index of the neighbor with the lowest index that has non-zero counts, or |S| + |P| if none applies.
- Types 10-13 are extended versions of type 0-3, each requiring further parameters of an integer default value and a list of weights, |S| integers enclosed in square brackets. Each action node is thus associated with an integer weight.
  - Type 10: Extended Sum. Each instance of an action in p's neighborhood being chosen contributes the weight of that action to the sum. These are added to the default value.
  - Type 11: Extended Existence: boolean for whether the extended sum is positive. The input default value and weights are required to be nonnegative.
  - Type 12: The weight of the neighbor with the highest index that has non-zero counts, or the default value if none applies.
  - Type 13: The weight of the neighbor with the lowest index that has non-zero counts, or the default value if none applies.

The following is an example of the signatures for an AGG with three action nodes and two function nodes:

```
2
10 0 [2 3 4]
```

8. The payoff function for each action node. So we have |S| subblocks of numbers. Payoff function for action s is a mapping from configurations to real numbers. Configurations are represented as a tuple of integers; the size of the tuple is the size of the neighborhood of s. Each configuration specifies the action counts for the neighbors of s, in the same order as the neighbor list of s.

   The first number of each subblock specifies the type of the payoff function. There are multiple ways of representing payoff functions; we (or other people) can extend the file format by defining new types of payoff functions. We define two basic types:

   Type 0
   The complete representation. The set of possible configurations can be derived from the action graph. This set of configurations can also be sorted in lexicographical order. So we can just specify the payoffs without explicitly giving the configurations. So we just need to give one row of real numbers, which correspond to payoffs for the ordered set of configurations.

   If action s is in multiple players' action sets (say players i, j), then it is possible that the set of possible configurations given s_i is different from the set of possible configurations given s_j. In such cases, we need to specify payoffs for the union of the sets of configurations (sorted in lexicographical order).

   Type 1
   The mapping representation, in which we specify the configurations and the corresponding payoffs. For the payoff function of action s, first give Delta_s, the number of elements in the mapping. Then follows Delta_s rows. In each row, first specify the configuration, which is a tuple of integers, enclosed by a pair of brackets "[" and "]", then the payoff. For example, the following specifies a payoff function of type 1, with two configurations:

   ```
   1 2
   [1 0] 2.5
   [1 1] -1.2
   ```

8.1.5 The Bayesian action graph game (.bagg) format

Bayesian action graph games (BAGGs) are a compact representation of Bayesian (i.e., incomplete-information) games. For more information on BAGGs, the following paper gives a detailed discussion.

> A.X. Jiang and K. Leyton-Brown, Bayesian Action-Graph Games. NIPS, 2010.

Each file in this format describes a BAGG. In order for the file to be recognized as BAGG by GAMBIT, the initial line of the file should be:

```
#BAGG
```

The rest of the file consists of the following sections, separated by whitespaces. Lines with starting '#' are treated as comments and are allowed between sections.

1. The number of Players, n.
2. The number of action nodes, |S|.
3. The number of function nodes, |P|.
4. The number of types for each player, as a row of n integers.
5. Type distribution for each player. The distributions are assumed to be independent. Each distribution is represented as a row of real numbers. The following example block gives the type distributions for a BAGG with two players and two types for each player:

   ```
   0.5 0.5
   0.2 0.8
   ```

6. Size of type-action set for each player's each type.
7. Type-action set for each player's each type. Each type-action set is represented as a row of integers in ascending order, which are indices of action nodes. Action nodes are indexed from 0 to |S|-1.
8. The action graph: same as in the AGG format.
9. types of functions: same as in the AGG format.
10. utility function for each action node: same as in the AGG format.

CHAPTER

NINE

BIBLIOGRAPHY

Note

To reference an entry in this bibliography, use the format `[key]_`, for example, `[Mye91]_` will link to the Myerson (1991) textbook entry.

9.1 Articles on computation of Nash equilibria

9.2 General game theory articles and texts

9.3 Textbooks and general reference

CHAPTER

TEN

DETAILED TABLE OF CONTENTS

BIBLIOGRAPHY

[BlaTur23] Bland, J. R. and Turocy, T. L. 2023, 'Quantal response equilibrium as a structural model for estimation: the missing manual', *SSRN Working Paper*, no. 4425515.

[Eav71] Eaves, B. C. 1971, 'The linear complementarity problem', *Management Science*, vol. 17, pp. 612-634.

[GovWil03] Govindan, S. and Wilson, R. 2003, 'A global Newton method to compute Nash equilibria', *Journal of Economic Theory*, vol. 110, no. 1, pp. 65-86.

[GovWil04] Govindan, S. and Wilson, R. 2004, 'Computing Nash equilibria by iterated polymatrix approximation', *Journal of Economic Dynamics and Control*, vol. 28, pp. 1229-1241.

[Jiang11] Jiang, A. X., Leyton-Brown, K. and Bhat, N. 2011, 'Action-graph games', *Games and Economic Behavior*, vol. 71, no. 1, pp. 141-173.

[KolMegSte94] Koller, D., Megiddo, N. and von Stengel, B. 1996, 'Efficient computation of equilibria for extensive two-person games', *Games and Economic Behavior*, vol. 14, pp. 247-259.

[LemHow64] Lemke, C. E. and Howson, J. T. 1964, 'Equilibrium points of bimatrix games', *Journal of the Society of Industrial and Applied Mathematics*, vol. 12, pp. 413-423.

[Man64] Mangasarian, O. 1964, 'Equilibrium points in bimatrix games', *Journal of the Society for Industrial and Applied Mathematics*, vol. 12, pp. 778-780.

[McK91] McKelvey, R. 1991, 'A Liapunov function for Nash equilibria', California Institute of Technology.

[McKMcL96] McKelvey, R. and McLennan, A. 1996, 'Computation of equilibria in finite games', in Amman, H., Kendrick, D. and Rust, J. (eds), *Handbook of Computational Economics*, Elsevier, pp. 87-142.

[PNS04] Porter, R., Nudelman, E. and Shoham, Y. 2004, 'Simple search methods for finding a Nash equilibrium', *Games and Economic Behavior*, pp. 664-669.

[Ros71] Rosenmuller, J. 1971, 'On a generalization of the Lemke-Howson algorithm to noncooperative n-person games', *SIAM Journal of Applied Mathematics*, vol. 21, pp. 73-79.

[Sha74] Shapley, L. 1974, 'A note on the Lemke-Howson algorithm', *Mathematical Programming Study*, vol. 1, pp. 175-189.

[Tur05] Turocy, T. L. 2005, 'A dynamic homotopy interpretation of the logistic quantal response equilibrium correspondence', *Games and Economic Behavior*, vol. 51, pp. 243-263.

[Tur10] Turocy, T. L. 2010, 'Using quantal response to compute Nash and sequential equilibria', *Economic Theory*, vol. 42, no. 1, pp. 255-269.

[VTH87] van der Laan, G., Talman, A. J. J. and van Der Heyden, L. 1987, 'Simplicial variable dimension algorithms for solving the nonlinear complementarity problem on a product of unit simplices using a general labelling', *Mathematics of Operations Research*, pp. 377-397.

[Wil71] Wilson, R. 1971, 'Computing equilibria of n-person games', *SIAM Applied Math*, vol. 21, pp. 80-87.

[Yam93] Yamamoto, Y. 1993, 'A path-following procedure to find a proper equilibrium of finite games', *International Journal of Game Theory*.

[Bag1995] Bagwell, K. 1995, 'Commitment and observability in games', *Games and Economic Behavior*, vol. 8, pp. 271-280.

[Harsanyi1967a] Harsanyi, J. 1967, 'Games of incomplete information played by Bayesian players I', *Management Science*, vol. 14, pp. 159-182.

[Harsanyi1967b] Harsanyi, J. 1967, 'Games of incomplete information played by Bayesian players II', *Management Science*, vol. 14, pp. 320-334.

[Harsanyi1968] Harsanyi, J. 1968, 'Games of incomplete information played by Bayesian players III', *Management Science*, vol. 14, pp. 486-502.

[KreWil82] Kreps, D. and Wilson, R. 1982, 'Sequential equilibria', *Econometrica*, vol. 50, pp. 863-894.

[Kre90] Kreps, D. 1990, *A Course in Microeconomic Theory*, Princeton University Press.

[McKPal95] McKelvey, R. and Palfrey, T. 1995, 'Quantal response equilibria for normal form games', *Games and Economic Behavior*, vol. 10, pp. 6-38.

[McKPal98] McKelvey, R. and Palfrey, T. 1998, 'Quantal response equilibria for extensive form games', *Experimental Economics*, vol. 1, pp. 9-41.

[Mye78] Myerson, R. 1978, 'Refinements of the Nash equilibrium concept', *International Journal of Game Theory*, vol. 7, pp. 73-80.

[Nas50] Nash, J. 1950, 'Equilibrium points in n-person games', *Proceedings of the National Academy of Sciences*, vol. 36, pp. 48-49.

[Och95] Ochs, J. 1995, 'Games with unique, mixed strategy equilibria: an experimental study', *Games and Economic Behavior*, vol. 10, pp. 202-217.

[Rei2008] Reiley, D. H., Urbancic, M. B. and Walker, M. 2008, 'Stripped-down poker: a classroom game with signaling and bluffing', *The Journal of Economic Education*, vol. 4, pp. 323-341.

[Sel75] Selten, R. 1975, 'Reexamination of the perfectness concept for equilibrium points in extensive games', *International Journal of Game Theory*, vol. 4, pp. 25-55.

[vanD83] van Damme, E. 1983, *Stability and Perfection of Nash Equilibria*, Springer-Verlag, Berlin.

[Mye91] Myerson, R. 1991, *Game Theory: Analysis of Conflict*, Harvard University Press.

[Wat13] Watson, J. 2013, *Strategy: An Introduction to Game Theory*, 3rd edn, W. W. Norton & Company.

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U