
Gambit Documentation

Release 16.2.0

The Gambit Project

Apr 16, 2024

CONTENTS

1	An overview of Gambit	3
2	Command-line tools	7
3	pygambit Python package	21
4	The graphical interface	87
5	Sample games	103
6	For developers: Building Gambit from source	105
7	Game representation formats	109
8	Bibliography	119
9	Detailed table of contents	121
	Bibliography	123
	Index	125

Gambit is a library of game theory software and tools for the construction and analysis of finite extensive and strategic games. Gambit is fully-cross platform, and is supported on Linux, Mac OS X, and Microsoft Windows.

Key features of Gambit include:

- A *graphical user interface*, which uses `wxWidgets` to provide a common interface with native look-and-feel across platforms.
- All equilibrium-computing algorithms are available as *command-line tools*, callable from scripts and other programs.
- A *Python API* for developing scripting applications.

Gambit is Free/Open Source software, released under the terms of the [GNU General Public License](#), Version 2.

We hope you will find Gambit useful for both teaching and research applications. If you do use Gambit in a class, or in a paper, we would like to hear about it. We are especially interested in finding out what you like about Gambit, and where you think improvements could be made.

If you are citing Gambit in a paper, we suggest a citation of the form:

Savani, Rahul and Turocy, Theodore L. (2024) Gambit: The package for computation in game theory, Version 16.2.0. <http://www.gambit-project.org>.

Replace the version number and year as appropriate if you use a different release.

Python user guide An introduction to using the `pygambit` package in Python.

User guide

Python API reference The complete reference to all the functionality of `pygambit`.

API documentation

Command-line tools All Gambit's methods for equilibrium computation are available via command-line programs.

Command-line tools

Graphical interface Gambit's graphical interface lets you interactively create, explore, and find equilibria of games.

The graphical interface

AN OVERVIEW OF GAMBIT

1.1 What is Gambit?

Gambit is a set of software tools for doing computation on finite, noncooperative games. These comprise a graphical interface for interactively building and analyzing general games in extensive or strategy form; a number of command-line tools for computing Nash equilibria and other solution concepts in games; and, a set of file formats for storing and communicating games to external tools.

1.2 A brief history of Gambit

The Gambit Project was founded in the mid-1980s by Richard McKelvey at the California Institute of Technology. The original implementation was written in BASIC, with a simple graphical interface. This code was ported to C around 1990 with the help of Bruce Bell, and was distributed publicly as version 0.13 in 1991 and 1992.

A major step in the evolution of Gambit took place with the awarding of the NSF grants in 1994, with McKelvey and Andrew McLennan as principal investigators, and Theodore Turocy as the head programmer. The grants sponsored a complete rewrite of Gambit in C++. The graphical interface was made portable across platforms through the use of the wxWidgets library (<http://www.wxwidgets.org>). Version 0.94 of Gambit was released in the late summer of 1994, version 0.96 followed in 1999, and version 0.97 in 2002. During this time, many students at Caltech and Minnesota contributed to the effort by programming, testing, and/or documenting. These include, alphabetically, Bruce Bell, Anand Chelian, Matthew Derer, Nelson Escobar, Ben Freeman, Eugene Grayver, Todd Kaplan, Geoff Matters, Brian Trotter, Michael Vanier, Roberto Weber, and Gary Wu.

Over the same period, Bernhard von Stengel, of the London School of Economics, made significant contributions in the implementation of the sequence form methods for two-player extensive games, and for contributing his “clique” code for identification of equilibrium components in two-player strategic games, as well as other advice regarding Gambit’s implementation and architecture.

Development since the mid-2000s has focused on two objectives. First, the graphical interface was reimplemented and modernized, with the goal of following good interaction design principles, especially in regards to easing the learning curve for users new to Gambit and new to game theory. Second, the internal architecture of Gambit was refactored to increase interoperability between the tools provided by Gambit and those written independently.

Gambit is proud to have participated in the Google Summer of Code program in the summers of 2011 and 2012 as a mentoring organization. The Python API, which became part of Gambit from Gambit 13, was developed during these summers, thanks in particular to the work of Stephen Kunath and Alessandro Andrioni.

1.3 Key features of Gambit

Gambit has a number of features useful both for the researcher and the instructor:

Interactive, cross-platform graphical interface. All Gambit features are available through the use of a graphical interface, which runs under multiple operating systems: Windows, various flavors of Unix (including Linux), and Mac OS X. The interface offers flexible methods for creating extensive and strategic games. It offers an interface for running algorithms to compute Nash equilibria, and for visualizing the resulting profiles on the game tree or table, as well as an interactive tool for analyzing the dominance structure of actions or strategies in the game. The interface is useful for the advanced researcher, but is intended to be accessible for students taking a first course in game theory as well.

Command-line tools for computing equilibria. More advanced applications often require extensive computing time and/or the ability to script computations. All algorithms in Gambit are packaged as individual, command-line programs, whose operation and output are configurable.

Extensibility and interoperability. The Gambit tools read and write file formats which are textual and documented, making them portable across systems and able to interact with external tools. It is therefore straightforward to extend the capabilities of Gambit by, for example, implementing a new method for computing equilibria, reimplementing an existing one more efficiently, or creating tools to programmatically create, manipulate, and transform games, or for econometric analysis on games.

1.4 Limitations of Gambit

Gambit has a few limitations that may be important in some applications. We outline them here.

Gambit is for finite games only. Because of the mathematical structure of finite games, it is possible to write many general-purpose routines for analyzing these games. Thus, Gambit can be used in a wide variety of applications of game theory. However, games that are not finite, that is, games in which players may choose from a continuum of actions, or in which players may have a continuum of types, do not admit the same general-purpose methods.

Gambit is for noncooperative game theory only. Gambit focuses on the branch of game theory in which the rules of the game are written down explicitly, and in which players choose their actions independently. Gambit's analytical tools center primarily around Nash equilibrium, and related concepts of bounded rationality such as quantal response equilibrium. Gambit does not at this time provide any representations of, or methods for, analyzing games written in cooperative form. (It should be noted that some problems in cooperative game theory do not suffer from the computational complexity that the Nash equilibrium problem does, and thus cooperative concepts could be an interesting future direction of development.)

Analyzing large games may become infeasible surprisingly quickly. While the specific formal complexity classes of computing Nash equilibria and related concepts are still an area of active research, it is clear that, in the typical case, the amount of time required to compute equilibria increases rapidly in the size of the game. In other words, it is quite easy to write down games which will take Gambit an unacceptably long amount time to compute the equilibria of. There are two ways to deal with this problem in practice. One way is to better identify good heuristic approaches for guiding the equilibrium computation process. Another way is to take advantage of known features of the game to guide the process. Both of these approaches are now becoming areas of active interest. While it will certainly not be possible to analyze every game that one would like to, it is hoped that Gambit will both contribute to these two areas of research, as well as make the resulting methods available to both students and practitioners.

1.5 Developers

The principal developers of Gambit are:

- [Theodore Turocy](#), University of East Anglia: director.
- Richard D. McKelvey, California Institute of Technology: project founder.
- Andrew McLennan, University of Queensland: co-PI during main development, developer and maintainer of polynomial-based algorithms for equilibrium computation.

Much of the development of the main Gambit codebase took place in 1994-1996, under a grant from the National Science Foundation to the California Institute of Technology and the University of Minnesota (McKelvey and McLennan, principal investigators).

Others contributing to the development and distribution of Gambit include:

- Bernhard von Stengel provided advice on implementation of sequence form code, and contributed clique code
- Eugene Grayver developed the first version of the graphical user interface.
- Gary Wu implemented an early scripting language interface for Gambit (since superseded by the Python API).
- Stephen Kunath and Alessandro Andrioni did extensive work to create the first release of the Python API.
- From Gambit 14, Gambit contains support for Action Graph Games [[Jiang11](#)]. This has been contributed by Navin Bhat, Albert Jiang, Kevin Leyton-Brown, and David Thompson, with funding support provided by a University Graduate Fellowship of the University of British Columbia, the NSERC Canada Graduate Scholarship, and a Google Research Award to Leyton-Brown.

1.6 Downloading Gambit

Gambit source code and built binaries can be downloaded from the project `GitHub repository releases section` <<https://github.com/gambitproject/gambit/releases>>.

Older versions of Gambit can be downloaded from <http://sourceforge.net/projects/gambit/files>.

1.7 Bug reports

In the first instance, bug reports or feature requests should be posted to the Gambit issue tracker, located at <http://github.com/gambitproject/gambit/issues>.

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.

COMMAND-LINE TOOLS

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 [McKMCL96].

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.

2.1 **`gambit-enumpure`**: Enumerate pure-strategy equilibria of a game

`gambit-enumpure` reads a game on standard input and searches for pure-strategy Nash equilibria.

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`

New 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 equilibrium profile computed.

`-A`

New in version 14.0.2.

Report agent form equilibria, that is, equilibria which consider only deviations at one information set. Only has an effect for extensive games, as strategic games have only one information set per player.

-P

By default, the program computes all pure-strategy Nash equilibria in an extensive game. This switch instructs the program to find only pure-strategy Nash equilibria which are subgame perfect. (This has no effect for strategic games, since there are no proper subgames of a strategic game.)

-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 16.2.0, Copyright (C) 1994-2024, 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 16.2.0, Copyright (C) 1994-2024, 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 16.2.0, Copyright (C) 1994-2024, 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
```

2.2 `gambit-enummixed`: Enumerate equilibria in a two-player game

`gambit-enummixed` reads a two-player game on standard input and 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 *`gambit-lcp`: Compute equilibria in a two-player game via linear complementarity*). It was shown by Shapley [Sha74] that there are equilibria not accessible via the method in *`gambit-lcp`: Compute equilibria in a two-player game via linear complementarity*, whereas the output of **`gambit-enummixed`** is guaranteed to return all the extreme points.

-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 equilibria 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-ennummixed e02.nfg
Compute Nash equilibria by enumerating extreme points
Gambit version 16.2.0, Copyright (C) 1994-2024, The Gambit Project
Enumeration code based on lrslib 4.2b,
Copyright (C) 1995-2005 by David Avis (avis@cs.mcgill.ca)
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-ennummixed -c e02.nfg
Compute Nash equilibria by enumerating extreme points
Gambit version 16.2.0, Copyright (C) 1994-2024, The Gambit Project
Enumeration code based on lrslib 4.2b,
Copyright (C) 1995-2005 by David Avis (avis@cs.mcgill.ca)
This is free software, distributed under the GNU GPL

NE,1,0,0,1,0
NE,1,0,0,1/2,1/2
```

(continues on next page)

(continued from previous page)

```
convex-1,1,0,0,1/2,1/2
convex-1,1,0,0,1,0
```

2.3 `gambit-lcp`: Compute equilibria in a two-player game via linear complementarity

gambit-lcp reads a two-player game on standard input and computes Nash equilibria 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 using the method of Lemke and Howson [LemHow64]. There exist strategic games for which some equilibria cannot be located by this method; 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 *gambit-enummixed: Enumerate equilibria in a two-player game* for a method which is guaranteed to find all the extreme points for a strategic game.

-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

New 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 equilibrium profile computed.

-P

By default, the program computes Nash equilibria in an extensive game. This switch instructs the program to find only equilibria which are subgame perfect. (This has no effect for strategic games, since there are no proper subgames of a strategic game.)

-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 16.2.0, Copyright (C) 1994-2024, The Gambit Project
```

(continues on next page)

(continued from previous page)

This is free software, distributed under the GNU GPL

NE,1,0,1/2,1/2,1/2,1/2

2.4 gambit-lp: Compute equilibria in a two-player constant-sum game via linear programming

gambit-lp reads a two-player constant-sum game on standard input and computes a Nash equilibrium by solving a linear program. The program uses the sequence form formulation of Koller, Megiddo, and von Stengel [KolMegSte94] for extensive games.

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.

-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

New 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 equilibrium profile computed.

-P

By default, the program computes Nash equilibria in an extensive game. This switch instructs the program to find only equilibria which are subgame perfect. (This has no effect for strategic games, since there are no proper subgames of a strategic game.)

-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 16.2.0, Copyright (C) 1994-2024, The Gambit Project
This is free software, distributed under the GNU GPL
```

(continues on next page)

NE, 1/3, 2/3, 1/3, 2/3

2.5 `gambit-liap`: Compute Nash equilibria using function minimization

gambit-liap reads a game on standard input and 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.

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 Lyapunov 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.

-d

Express all output using decimal representations with the specified number of digits.

-n

Specify the number of starting points to randomly generate.

-i

New in version 16.1.1.

Specify the maximum number of iterations in function minimization (default is 1000).

-m

New in version 16.2.0.

Specify the maximum regret criterion for acceptance as an approximate Nash equilibrium (default is 1e-4). See [Acceptance criteria for Nash equilibria](#) 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 16.2.0, Copyright (C) 1994-2024, The Gambit Project
This is free software, distributed under the GNU GPL

NE, 0.998701, 0.000229, 0.001070, 0.618833, 0.381167
```

2.6 gambit-simpdiv: Compute equilibria via simplicial subdivision

`gambit-simpdiv` reads a game on standard input and 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.

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

New in version 16.2.0.

Specify the maximum regret criterion for acceptance as an approximate Nash equilibrium (default is 1e-8). See [Acceptance criteria for Nash equilibria](#) for interpretation and guidance.

-d DECIMALS

New 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 16.2.0, Copyright (C) 1994-2024, The Gambit Project
This is free software, distributed under the GNU GPL

NE,1,0,0,1,0
```

2.7 `gambit-logit`: Compute quantal response equilibria

gambit-logit reads a game on standard input and 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 option `-s` sets the 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 option `-a` sets the maximum acceleration (or deceleration). 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.

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 than .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

New in version 16.2.0.

Specify the maximum regret criterion for acceptance as an approximate Nash equilibrium (default is 1e-8). See [Acceptance criteria for Nash equilibria](#) for interpretation and guidance.

-l

While tracing, compute the logit equilibrium points with parameter LAMBDA accurately.

-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 16.2.0, Copyright (C) 1994-2024, 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
```

(continues on next page)

(continued from previous page)

```
979103.323545,1,0,2.54469e-11,0.500012,0.499988
1077013.665501,1,0,2.11883e-11,0.500011,0.499989
```

2.8 `gambit-gnm`: Compute Nash equilibria in a strategic game using a global Newton method

gambit-gnm reads a game on standard input and 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.

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

New 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

New 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

New in version 16.2.0.

Specifies the maximum number of iterations in a local Newton method step. Default is 10.

-c STEPS

New 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 16.2.0, Copyright (C) 1994-2024, 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`: Compute Nash equilibria in a strategic game using iterated polymatrix approximation.

2.9 `gambit-ipa`: Compute Nash equilibria in a strategic game using iterated polymatrix approximation

`gambit-ipa` reads a game on standard input and 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.

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 16.2.0, Copyright (C) 1994-2024, 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: Compute Nash equilibria in a strategic game using a global Newton method.

2.10 gambit-convert: Convert games among various representations

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 16.2.0, Copyright (C) 1994-2024, The Gambit Project
This is free software, distributed under the GNU GPL

<center><h1>Two person 2 x 2 game with unique mixed equilibrium</h1></center>
<table><tr><td></td><td align=center><b>1</b></td><td
align=center><b>2</b></td></tr><tr><td align=center><b>1</b></td><td
align=center>2,0</td><td align=center>0,1</td></tr><tr><td
align=center><b>2</b></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 16.2.0, Copyright (C) 1994-2024, 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}
```


PYGAMBIT PYTHON PACKAGE

Gambit provides a Python package, `pygambit`, which provides access to Gambit's features. `pygambit` is available on PyPI (<https://pypi.org/project/pygambit/>), and can be installed via `pip`.

3.1 User guide

3.1.1 Example: One-shot trust game with binary actions

[Kre90] introduced a game commonly referred to as the **trust game**. We will build a one-shot version of this game using `pygambit`'s game transformation operations.

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.

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

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

In [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:

```
In [3]: g.root
Out[3]: Node(game=Game(title='One-shot trust game, after Kreps (1990)'), path=[])

In [4]: g.root.children
Out[4]: NodeChildren(parent=Node(game=Game(title='One-shot trust game, after Kreps (1990)
↪'), path=[]))
```

To extend a game from an existing terminal node, use `Game.append_move()`:

```
In [5]: g.append_move(g.root, "Buyer", ["Trust", "Not trust"])

In [6]: g.root.children
Out[6]: NodeChildren(parent=Node(game=Game(title='One-shot trust game, after Kreps (1990)
↪'), path=[]))
```

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

```
In [7]: g.append_move(g.root.children[0], "Seller", ["Honor", "Abuse"])
```

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:

```
In [8]: g.set_outcome(g.root.children[0].children[0], g.add_outcome([1, 1], label=
↳ "Trustworthy"))
```

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

```
In [9]: g.set_outcome(g.root.children[0].children[1], g.add_outcome([-1, 2], label=
↳ "Untrustworthy"))
```

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

```
In [10]: g.set_outcome(g.root.children[1], g.add_outcome([0, 0], label="Opt-out"))
```

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.

3.1.2 Example: A one-card poker game with private information

To illustrate games in extensive form, [Mye91] presents a one-card poker game. A version of this game also appears in [RUW08], as a classroom game under the name “stripped-down poker”. This is perhaps the simplest interesting game with imperfect information.

In our version of the game, there are two players, **Alice** and **Bob**. There is a deck of cards, with equal numbers of **King** and **Queen** cards. The game begins with each player putting \$1 in the pot. One card is dealt at random to Alice; Alice observes her card but Bob does not. After Alice observes her card, she can choose either to **Raise** or to **Fold**. If she chooses to Fold, Bob wins the pot and the game ends. If she chooses to Raise, she adds another \$1 to the pot. Bob then chooses either to **Meet** or **Pass**. If he chooses to Pass, Alice wins the pot and the game ends. If he chooses to Meet, 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.

We can build this game using the following script:

```
g = gbt.Game.new_tree(players=["Alice", "Bob"],
                        title="One card poker game, after Myerson (1991)")
g.append_move(g.root, g.players.chance, ["King", "Queen"])
for node in g.root.children:
    g.append_move(node, "Alice", ["Raise", "Fold"])
g.append_move(g.root.children[0].children[0], "Bob", ["Meet", "Pass"])
g.append_infoset(g.root.children[1].children[0],
                 g.root.children[0].children[0].infoset)
alice_winsbig = g.add_outcome([2, -2], label="Alice wins big")
alice_wins = g.add_outcome([1, -1], label="Alice wins")
bob_winsbig = g.add_outcome([-2, 2], label="Bob wins big")
bob_wins = g.add_outcome([-1, 1], label="Bob wins")
g.set_outcome(g.root.children[0].children[0].children[0], alice_winsbig)
g.set_outcome(g.root.children[0].children[0].children[1], alice_wins)
g.set_outcome(g.root.children[0].children[1], bob_wins)
g.set_outcome(g.root.children[1].children[0].children[0], bob_winsbig)
```

(continues on next page)

(continued from previous page)

```
g.set_outcome(g.root.children[1].children[0].children[1], alice_wins)
g.set_outcome(g.root.children[1].children[1], bob_wins)
```

All extensive games have a chance (or nature) player, accessible as `.Game.players.chance`. Moves belonging to the chance player can be added in the same way as to personal players. At any new move created for the chance player, the action probabilities default to uniform randomization over the actions at the move.

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. The loop:

```
for node in g.root.children:
    g.append_move(node, "Alice", ["Raise", "Fold"])
```

causes each of the newly-appended moves to be in new information sets. In contrast, Bob does not know Alice's card, and therefore cannot distinguish between the two nodes at which he has the decision. This is implemented in the following lines:

```
g.append_move(g.root.children[0].children[0], "Bob", ["Meet", "Pass"])
g.append_infoset(g.root.children[1].children[0],
                 g.root.children[0].children[0].infoset)
```

The call `Game.append_infoset()` adds a move at a terminal node as part of an existing information set (represented in pygambit as an *Infoset*).

3.1.3 Building a strategic game

Games in strategic form, also referred to as normal form, are represented solely by a collection of payoff tables, one per player. The most direct way to create a strategic 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.

For example, to create a standard prisoner's dilemma game in which the cooperative payoff is 8, the betrayal payoff is 10, the sucker payoff is 2, and the noncooperative payoff is 5:

```
In [11]: import numpy as np

In [12]: m = np.array([[8, 2], [10, 5]])

In [13]: g = gbt.Game.from_arrays(m, np.transpose(m))

In [14]: g
Out[14]: Game(title='Untitled strategic game')
```

The arrays passed to `Game.from_arrays()` are all indexed in the same sense, that is, the top level index is the choice of the first player, the second level index of the second player, and so on. Therefore, to create a two-player symmetric game, as in this example, the payoff matrix for the second player is transposed before passing to `Game.from_arrays()`.

3.1.4 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, we consider a trivial game which just has one move for the chance player:

```
In [15]: import pygambit as gbt
In [16]: g = gbt.Game.new_tree()
In [17]: g.append_move(g.root, g.players.chance, ["a", "b", "c"])
In [18]: [act.prob for act in g.root.infoset.actions]
Out[18]: [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:

```
In [19]: g.set_chance_probs(g.root.infoset,
.....:                     [gbt.Rational(1, 4), gbt.Rational(1, 2), gbt.Rational(1, 4)])
.....:
In [20]: [act.prob for act in g.root.infoset.actions]
Out[20]: [Rational(1, 4), Rational(1, 2), Rational(1, 4)]
```

They can also be explicitly specified as decimal numbers:

```
In [21]: g.set_chance_probs(g.root.infoset,
.....:                     [gbt.Decimal(".25"), gbt.Decimal(".50"), gbt.Decimal(".25")])
.....:
In [22]: [act.prob for act in g.root.infoset.actions]
Out[22]: [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:

```
In [23]: g.set_chance_probs(g.root.infoset, ["1/4", "1/2", "1/4"])
In [24]: [act.prob for act in g.root.infoset.actions]
Out[24]: [Rational(1, 4), Rational(1, 2), Rational(1, 4)]
In [25]: g.set_chance_probs(g.root.infoset, [".25", ".50", ".25"])
In [26]: [act.prob for act in g.root.infoset.actions]
Out[26]: [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,

```
In [27]: g.set_chance_probs(g.root.infoset, [.25, .50, .25])
```

```
In [28]: [act.prob for act in g.root.infoset.actions]
```

```
Out[28]: [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

```
In [29]: g.set_chance_probs(g.root.infoset, [1/3, 1/3, 1/3])
```

```
-----
ValueError                                Traceback (most recent call last)
Cell In[29], line 1
----> 1 g.set_chance_probs(g.root.infoset, [1/3, 1/3, 1/3])

File src/pygambit/game.pxi:632, in pygambit.gambit.Game.set_chance_probs()

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,

```
In [30]: 1/3 + 1/3 + 1/3
```

```
Out[30]: 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

```
In [31]: gbt.Decimal(str(1/3))
```

```
Out[31]: Decimal('0.3333333333333333')
```

and the sum-to-one check then fails because

```
In [32]: gbt.Decimal(str(1/3)) + gbt.Decimal(str(1/3)) + gbt.Decimal(str(1/3))
```

```
Out[32]: 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.

3.1.5 Reading a game from a file

Games stored in existing Gambit savefiles can be loaded using *Game.read_game()*:

```
In [33]: g = gbt.Game.read_game("e02.nfg")
```

```
In [34]: g
```

```
Out[34]: Game(title='Selten (IJGT, 75), Figure 2, normal form')
```

3.1.6 Computing Nash equilibria

Interfaces to algorithms for computing Nash equilibria are provided in `pygambit.nash`.

Method	Python function
<code>gambit-enumpure</code>	<code>pygambit.nash.enumpure_solve()</code>
<code>gambit-enummixed</code>	<code>pygambit.nash.enummixed_solve()</code>
<code>gambit-lp</code>	<code>pygambit.nash.lp_solve()</code>
<code>gambit-lcp</code>	<code>pygambit.nash.lcp_solve()</code>
<code>gambit-liap</code>	<code>pygambit.nash.liap_solve()</code>
<code>gambit-logit</code>	<code>pygambit.nash.logit_solve()</code>
<code>gambit-simpdiv</code>	<code>pygambit.nash.simpdiv_solve()</code>
<code>gambit-ipa</code>	<code>pygambit.nash.ipa_solve()</code>
<code>gambit-gnm</code>	<code>pygambit.nash.gnm_solve()</code>

We take as an example the *one-card poker game*. This is a two-player, constant sum game, and so all of the equilibrium-finding methods can be applied to it.

For two-player games, `lcp_solve()` can compute Nash equilibria directly using the extensive representation. Assuming that `g` refers to the game

```
In [35]: result = gbt.nash.lcp_solve(g)

In [36]: result
Out[36]: 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': 0, 'max_depth': 0})

In [37]: len(result.equilibria)
Out[37]: 1
```

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 mixed behavior profiles. A mixed behavior profile specifies, for each information set, the probability distribution over actions at that information set. Indexing a `MixedBehaviorProfile` by a player gives a `MixedBehavior`, which specifies probability distributions at each of the player's information sets:

```
In [38]: eqm = result.equilibria[0]

In [39]: eqm["Alice"]
Out[39]: [[Rational(1, 1), Rational(0, 1)], [Rational(1, 3), Rational(2, 3)]]
```

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

```
In [40]: for infoset, mixed_action in eqm["Alice"].mixed_actions():
.....:     print(infoset)
.....:     print(mixed_action)
.....:
Infoset(player=Player(game=Game(title='One card poker game, after Myerson (1991)'),
↳ label='Alice'), number=0)
```

(continues on next page)

(continued from previous page)

```
[Rational(1, 1), Rational(0, 1)]
Infoset(player=Player(game=Game(title='One card poker game, after Myerson (1991)'),
↳label='Alice'), number=1)
[Rational(1, 3), Rational(2, 3)]
```

So we could extract Alice’s probabilities of raising at her respective information sets like this:

```
In [41]: {infoset: mixed_action["Raise"] for infoset, mixed_action in eqm["Alice"].mixed_
↳actions()}
Out[41]:
{Infoset(player=Player(game=Game(title='One card poker game, after Myerson (1991)'),
↳label='Alice'), number=0): Rational(1, 1),
  Infoset(player=Player(game=Game(title='One card poker game, after Myerson (1991)'),
↳label='Alice'), number=1): Rational(1, 3)}
```

In larger games, labels may not always be the most convenient way to refer to specific actions. We can also index profiles directly with [Action](#) objects. So an alternative way to extract the probabilities of playing “Raise” would be by iterating Alice’s list of actions:

```
In [42]: {action.infoset: eqm[action] for action in g.players["Alice"].actions if action.
↳label == "Raise"}
Out[42]:
{Infoset(player=Player(game=Game(title='One card poker game, after Myerson (1991)'),
↳label='Alice'), number=0): Rational(1, 1),
  Infoset(player=Player(game=Game(title='One card poker game, after Myerson (1991)'),
↳label='Alice'), number=1): Rational(1, 3)}
```

Looking at Bob’s strategy,

```
In [43]: eqm["Bob"]
Out[43]: [[Rational(2, 3), Rational(1, 3)]]
```

Bob meets Alice’s raise two-thirds of the time. The label “Raise” 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 “Meet” in the game, we can extract the probability that Bob plays “Meet” by:

```
In [44]: eqm["Bob"]["Meet"]
Out[44]: Rational(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:

```
In [45]: eqm["Meet"]
Out[45]: Rational(2, 3)
```

Because this is an equilibrium, the fact that Bob randomizes at his information set must mean he is indifferent between the two actions at his information set. [MixedBehaviorProfile.action_value\(\)](#) returns the expected payoff of taking an action, conditional on reaching that action’s information set:

```
In [46]: {action: eqm.action_value(action) for action in g.players["Bob"].infosets[0].
↳actions}
Out[46]:
```

(continues on next page)

(continued from previous page)

```
{Action(infoset=Infoset(player=Player(game=Game(title='One card poker game, after_
↳Myerson (1991)'), label='Bob'), number=0), label='Meet'): Rational(-1, 1),
  Action(infoset=Infoset(player=Player(game=Game(title='One card poker game, after_
↳Myerson (1991)'), label='Bob'), number=0), label='Pass'): Rational(-1, 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:

```
In [47]: {node: eqm.belief(node) for node in g.players["Bob"].infosets[0].members}
Out[47]:
{Node(game=Game(title='One card poker game, after Myerson (1991)'), path=[0, 0]):_
↳Rational(3, 4),
  Node(game=Game(title='One card poker game, after Myerson (1991)'), path=[0, 1]):_
↳Rational(1, 4)}
```

Bob believes that, conditional on Alice raising, there's a 75% chance that she has the king; therefore, the expected payoff to meeting is in fact -1 as computed. *MixedBehaviorProfile.infoset_prob()* returns the probability that an information set is reached:

```
In [48]: eqm.infoset_prob(g.players["Bob"].infosets[0])
Out[48]: Rational(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()*.

```
In [49]: {node: eqm.node_value("Bob", node) for node in g.players["Bob"].infosets[0].
↳members}
Out[49]:
{Node(game=Game(title='One card poker game, after Myerson (1991)'), path=[0, 0]):_
↳Rational(-5, 3),
  Node(game=Game(title='One card poker game, after Myerson (1991)'), path=[0, 1]):_
↳Rational(1, 1)}
```

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

```
In [50]: eqm.payoff("Alice")
Out[50]: Rational(1, 3)

In [51]: eqm.payoff("Bob")
Out[51]: Rational(-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

```
In [52]: [s.label for s in g.players["Alice"].strategies]
Out[52]: ['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. We can therefore apply a method which operates on a strategic game to any game with an extensive representation

```
In [53]: result = gbt.nash.gnm_solve(g)
```

```
In [54]: result
```

```
Out[54]: NashComputationResult(method='gnm', rational=False, use_strategic=True,
↳ equilibria=[[0.33333333333866677, 0.6666666666613335, 0.0, 0.0], [0.6666666666559997,
↳ 0.3333333333440004]], 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*, and specify, for each player, a probability distribution over that player's strategies. Indexing a *MixedStrategyProfile* by a player gives the probability distribution over that player's strategies only.

```
In [55]: eqm = result.equilibria[0]
```

```
In [56]: eqm["Alice"]
```

```
Out[56]: [0.33333333333866677, 0.6666666666613335, 0.0, 0.0]
```

```
In [57]: eqm["Bob"]
```

```
Out[57]: [0.6666666666559997, 0.3333333333440004]
```

The expected payoff to a strategy is provided by *MixedStrategyProfile.strategy_value()*:

```
In [58]: {strategy: eqm.strategy_value(strategy) for strategy in g.players["Alice"].
↳ strategies}
```

```
Out[58]:
```

```
{Strategy(player=Player(game=Game(title='One card poker game, after Myerson (1991)'),
↳ label='Alice'), label='11'): 0.33333333334400045,
  Strategy(player=Player(game=Game(title='One card poker game, after Myerson (1991)'),
↳ label='Alice'), label='12'): 0.33333333332799997,
  Strategy(player=Player(game=Game(title='One card poker game, after Myerson (1991)'),
↳ label='Alice'), label='21'): -0.9999999999839995,
  Strategy(player=Player(game=Game(title='One card poker game, after Myerson (1991)'),
↳ label='Alice'), label='22'): -1.0}
```

```
In [59]: {strategy: eqm.strategy_value(strategy) for strategy in g.players["Bob"].
↳ strategies}
```

```
Out[59]:
```

```
{Strategy(player=Player(game=Game(title='One card poker game, after Myerson (1991)'),
↳ label='Bob'), label='1'): -0.33333333333066656,
  Strategy(player=Player(game=Game(title='One card poker game, after Myerson (1991)'),
↳ label='Bob'), label='2'): -0.3333333333386667}
```

The overall expected payoff to a player is returned by *MixedStrategyProfile.payoff()*:

```
In [60]: eqm.payoff("Alice")
```

```
Out[60]: 0.3333333333333354
```

(continues on next page)

(continued from previous page)

```
In [61]: eqm.payoff("Bob")
Out[61]: -0.3333333333333354
```

When a game has an extensive representation, we can convert freely between *MixedStrategyProfile* and the corresponding *MixedBehaviorProfile* representation of the same strategies using *MixedStrategyProfile.as_behavior()* and *MixedBehaviorProfile.as_strategy()*.

```
In [62]: eqm.as_behavior()
Out[62]: [[[1.0, 0.0], [0.3333333333386667, 0.6666666666613333]], [[0.6666666666559997,
↪ 0.3333333333440004]]]

In [63]: eqm.as_behavior().as_strategy()
Out[63]: [[0.3333333333386667, 0.6666666666613333, 0.0, 0.0], [0.6666666666559997, 0.
↪ 3333333333440004]]
```

3.1.7 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 .

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 the standard one-card poker game using *logit_solve()*. The range of the payoffs in this game is 4 (from +2 to -2).

```
In [64]: g = gbt.Game.read_game("poker.efg")

In [65]: g.max_payoff, g.min_payoff
Out[65]: (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 10^{-8} :

```
In [66]: result = gbt.nash.logit_solve(g, maxregret=1e-8)

In [67]: result.equilibria
Out[67]: [[[[1.0, 0.0], [0.33333338649882943, 0.6666666135011707]], [[0.6666667065407631,
↪ 0.3333332934592369]]]]

In [68]: result.equilibria[0].max_regret()
Out[68]: 3.987411578698641e-08
```

The value of *MixedBehaviorProfile.max_regret()* of the computed profile exceeds 10^{-8} measured in payoffs of the game. However, when considered relative to the scale of the game's payoffs, we see it is less than 10^{-8} of the payoff range, as requested:

```
In [69]: result.equilibria[0].max_regret() / (g.max_payoff - g.min_payoff)
Out[69]: 9.968528946746602e-09
```

In general, for globally-convergent methods especially, there is a tradeoff between precision and running time. Some methods may be slow to converge on some games, and it may be useful instead to get a more coarse approximation to an equilibrium. We could instead ask only for an ε -equilibrium with a (scaled) ε of no more than 10^{-4} :

```
In [70]: result = gbt.nash.logit_solve(g, maxregret=1e-4)

In [71]: result.equilibria[0]
Out[71]: [[[1.0, 0.0], [0.3338351656285656, 0.666164834417892]], [[0.6670407651644306, 0.
↪ 3329592348608147]]]

In [72]: result.equilibria[0].max_regret()
Out[72]: 0.00037581039824041707

In [73]: result.equilibria[0].max_regret() / (g.max_payoff - g.min_payoff)
Out[73]: 9.395259956010427e-05
```

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()`.

```
In [74]: result = gbt.nash.liap_solve(g.mixed_behavior_profile(), maxregret=1.0e-4)

In [75]: result.equilibria[0]
Out[75]: [[[0.9999979852809777, 2.0147190221904606e-06], [0.3333567614695495, 0.
↪ 6666432385304505]], [[0.6668870457043463, 0.33311295429565374]]]

In [76]: result.equilibria[0].max_regret()
Out[76]: 0.00022039452688993322

In [77]: result.equilibria[0].max_regret() / (g.max_payoff - g.min_payoff)
Out[77]: 5.5098631722483304e-05
```

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

```
In [78]: for outcome in g.outcomes:
.....:     outcome["Alice"] = outcome["Alice"] * 2
.....:     outcome["Bob"] = outcome["Bob"] * 2
.....:

In [79]: result = gbt.nash.liap_solve(g.mixed_behavior_profile(), maxregret=1.0e-4)

In [80]: result.equilibria[0]
Out[80]: [[[0.9999979852809777, 2.0147190221904606e-06], [0.3333567614695495, 0.
↪ 6666432385304505]], [[0.6668870457043463, 0.33311295429565374]]]

In [81]: result.equilibria[0].max_regret()
Out[81]: 0.00044078905377986644

In [82]: result.equilibria[0].max_regret() / (g.max_payoff - g.min_payoff)
Out[82]: 5.5098631722483304e-05
```

3.1.8 Generating starting points for algorithms

Some methods for computation of Nash equilibria take as an initial condition a *MixedStrategyProfile* or *MixedBehaviorProfile* which is used as a starting point. The equilibria found will depend on which starting point is selected. To facilitate generating starting points, *Game* provides methods *Game.random_strategy_profile()* and *Game.random_behavior_profile()*, to generate profiles which are drawn from the uniform distribution on the product of simplices.

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 equilibria in games of this size.

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.

```
In [83]: g = gbt.Game.read_game("2x2x2.nfg")

In [84]: gbt.nash.liap_solve(g.mixed_strategy_profile())
Out[84]: NashComputationResult(method='liap', rational=False, use_strategic=True,
→ equilibria=[[0.40000000330601743, 0.59999999669398257], [0.499999791913678, 0.
→ 50000002080863221], [0.3333334438355959, 0.6666665561644041]], parameters={'start':
→ [[0.5, 0.5], [0.5, 0.5], [0.5, 0.5]], 'maxregret': 0.0001, 'maxiter': 1000})
```

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.

```
In [85]: gbt.nash.liap_solve(g.mixed_strategy_profile([[.9, .1], [.9, .1], [.9, .1]]))
Out[85]: NashComputationResult(method='liap', rational=False, use_strategic=True,
→ equilibria=[[0.9999999952144591, 4.785540892853837e-09], [0.9999999998451463, 1.
→ 5485358776552024e-10], [0.9999999988047714, 1.1952285809728638e-09]], parameters={
→ 'start': [[0.9, 0.1], [0.9, 0.1], [0.9, 0.1]], 'maxregret': 0.0001, 'maxiter': 1000})
```

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

```
In [86]: gbt.nash.liap_solve(g.random_strategy_profile())
Out[86]: NashComputationResult(method='liap', rational=False, use_strategic=True,
→ equilibria=[[0.40000006115393901, 0.59999938846061], [0.49999873824756796, 0.
→ 50000012617524321], [0.3333336637527727, 0.6666663362472274]], parameters={'start':
→ [[0.5279265148370416, 0.4720734851629584], [0.39513832089736656, 0.6048616791026334],
→ [0.5806139651431909, 0.4193860348568091]], 'maxregret': 0.0001, 'maxiter': 1000})
```

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.

```
In [87]: import numpy as np

In [88]: gen = np.random.default_rng(seed=1234567890)

In [89]: p1 = g.random_strategy_profile(gen=gen)

In [90]: p1
```

(continues on next page)

(continued from previous page)

```
Out[90]: [[0.8359110871883912, 0.16408891281160876], [0.7422931336795922, 0.
↳ 25770686632040796], [0.977833473596193, 0.02216652640380706]]
```

```
In [91]: gen = np.random.default_rng(seed=1234567890)
```

```
In [92]: p2 = g.random_strategy_profile(gen=gen)
```

```
In [93]: p2
```

```
Out[93]: [[0.8359110871883912, 0.16408891281160876], [0.7422931336795922, 0.
↳ 25770686632040796], [0.977833473596193, 0.02216652640380706]]
```

```
In [94]: p1 == p2
```

```
Out[94]: True
```

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`.

```
In [95]: gen = np.random.default_rng(seed=1234567890)
```

```
In [96]: g.random_strategy_profile(denom=10, gen=gen)
```

```
Out[96]: [[Rational(1, 2), Rational(1, 2)], [Rational(7, 10), Rational(3, 10)],
↳ [Rational(0, 1), Rational(1, 1)]]
```

```
In [97]: g.random_strategy_profile(denom=10, gen=gen)
```

```
Out[97]: [[Rational(1, 10), Rational(9, 10)], [Rational(3, 5), Rational(2, 5)],
↳ [Rational(3, 5), Rational(2, 5)]]
```

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

```
In [98]: gbt.nash.simpdiv_solve(g.random_strategy_profile(denom=10, gen=gen))
```

```
Out[98]: NashComputationResult(method='simpdiv', rational=True, use_strategic=True,
↳ equilibria=[[Rational(1, 1), Rational(0, 1)], [Rational(1, 1), Rational(0, 1)],
↳ [Rational(1, 1), Rational(0, 1)]]], parameters={'start': [[Rational(7, 10), Rational(3,
↳ 10)], [Rational(4, 5), Rational(1, 5)], [Rational(0, 1), Rational(1, 1)]], 'maxregret
↳ ': Rational(1, 100000000), 'refine': 2, 'leash': None})
```

```
In [99]: gbt.nash.simpdiv_solve(g.random_strategy_profile(denom=10, gen=gen))
```

```
Out[99]: NashComputationResult(method='simpdiv', rational=True, use_strategic=True,
↳ equilibria=[[Rational(1, 1), Rational(0, 1)], [Rational(0, 1), Rational(1, 1)],
↳ [Rational(0, 1), Rational(1, 1)]]], parameters={'start': [[Rational(4, 5), Rational(1,
↳ 5)], [Rational(1, 5), Rational(4, 5)], [Rational(0, 1), Rational(1, 1)]], 'maxregret':
↳ Rational(1, 100000000), 'refine': 2, 'leash': None})
```

```
In [100]: gbt.nash.simpdiv_solve(g.random_strategy_profile(denom=10, gen=gen))
```

```
Out[100]: NashComputationResult(method='simpdiv', rational=True, use_strategic=True,
↳ equilibria=[[Rational(1, 1), Rational(0, 1)], [Rational(1, 1), Rational(0, 1)],
```

(continues on next page)

(continued from previous page)

```

↪[Rational(1, 1), Rational(0, 1)]]], parameters={'start': [[Rational(1, 2), Rational(1, 2)],
↪[Rational(1, 1), Rational(0, 1)], [Rational(1, 2), Rational(1, 2)]], 'maxregret':
↪Rational(1, 100000000), 'refine': 2, 'leash': None})

```

3.1.9 Estimating quantal response equilibria

Alongside computing quantal response equilibria, Gambit can also perform maximum likelihood estimation, computing the QRE which best fits an empirical distribution of play.

As an example we consider an asymmetric matching pennies game studied in [Och95], analysed in [McKPal95] using QRE.

```

In [101]: 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)"
.....: )
.....:

In [102]: data = g.mixed_strategy_profile([[128*0.527, 128*(1-0.527)], [128*0.366,
↪128*(1-0.366)]])

```

Estimation of QRE is done using `fit_fixedpoint()`.

```

In [103]: fit = gbt.qre.fit_fixedpoint(data)

```

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 likelihood-ratio tests.¹

```

In [104]: print(fit.lam)
1.8456097536855864

In [105]: print(fit.profile)
[[0.615651314427859, 0.3843486855721409], [0.3832909400456291, 0.616709059954371]]

In [106]: print(fit.log_like)
-174.76453191087444

```

¹ The log-likelihoods quoted in [McKPal95] are exactly a factor of 10 larger than those obtained by replicating the calculation.

3.2 API documentation

3.2.1 Representation of games

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

<code>add_action(infoset[, before])</code>	Add an action at the information set <i>infoset</i> .
<code>add_outcome([payoffs, label])</code>	Add a new outcome to the game.
<code>add_player([label])</code>	Add a new player to the game.
<code>add_strategy(player[, label])</code>	Add a new strategy to the set of strategies for <i>player</i> .
<code>append_infoset(nodes, infoset)</code>	Add a move in information set <i>infoset</i> at terminal <i>nodes</i> .
<code>append_move(nodes, player, actions)</code>	Add a move for <i>player</i> at terminal <i>nodes</i> .
<code>copy_tree(src, dest)</code>	Copy the subtree rooted at 'src' to 'dest'.
<code>delete_action(action)</code>	Deletes <i>action</i> from its information set.
<code>delete_outcome(outcome)</code>	Delete an outcome from the game.
<code>delete_parent(node)</code>	Delete the parent node of <i>node</i> .
<code>delete_strategy(strategy)</code>	Delete <i>strategy</i> from the game.
<code>delete_tree(node)</code>	Truncate the game tree at <i>node</i> , deleting the subtree beneath it.
<code>from_arrays(*arrays[, title])</code>	Create a new Game with a strategic representation.
<code>from_dict(payoffs[, title])</code>	Create a new Game with a strategic representation.
<code>insert_infoset(node, infoset)</code>	Insert a move in information set <i>infoset</i> prior to the node <i>node</i> .
<code>insert_move(node, player, actions)</code>	Insert a move for <i>player</i> prior to the node <i>node</i> , with <i>actions</i> actions.
<code>leave_infoset(node)</code>	Remove this node from its information set.
<code>mixed_behavior_profile([data, rational])</code>	Create a mixed behavior profile over the game.
<code>mixed_strategy_profile([data, rational])</code>	Create a mixed strategy profile over the game.
<code>move_tree(src, dest)</code>	Move the subtree rooted at 'src' to 'dest'.
<code>new_table(dim[, title])</code>	Create a new Game with a strategic representation.
<code>new_tree([players, title])</code>	Create a new Game consisting of a trivial game tree, with one node, which is both root and terminal.

continues on next page

Table 1 – continued from previous page

<code>nodes([subtree])</code>	Return a list of nodes in the game tree.
<code>parse_game(text)</code>	Construct a game from its serialised representation in a string
<code>random_behavior_profile([denom, gen])</code>	Create a <i>MixedBehaviorProfile</i> on the game, with probabilities drawn from the uniform distribution over the set of mixed behavior profiles.
<code>random_strategy_profile([denom, gen])</code>	Create a <i>MixedStrategy</i> on the game, with probabilities drawn from the uniform distribution over the set of mixed strategy profiles.
<code>read_game(filepath)</code>	Construct a game from its serialised representation in a file.
<code>reveal(infoset, player)</code>	Reveals the move made at <i>infoset</i> to <i>player</i> .
<code>set_chance_probs(infoset, probs)</code>	Set the action probabilities at chance information set <i>infoset</i> .
<code>set_infoset(node, infoset)</code>	Place <i>node</i> in the information set <i>infoset</i> .
<code>set_outcome(node, outcome)</code>	Set <i>outcome</i> to be the outcome at <i>node</i> .
<code>set_player(infoset, player)</code>	Set the player at an information set.
<code>support_profile()</code>	
<code>write([format])</code>	Produce a serialization of the game.

Attributes

<code>actions</code>	The set of actions available in the game.
<code>comment</code>	Get or set the comment of the game.
<code>contingencies</code>	An iterator over the contingencies in the game.
<code>infosets</code>	The set of information sets in the game.
<code>is_const_sum</code>	Whether the game is constant sum.
<code>is_perfect_recall</code>	Whether the game is perfect recall.
<code>is_tree</code>	Return whether a game has a tree-based representation.
<code>max_payoff</code>	The maximum payoff in the game.
<code>min_payoff</code>	The minimum payoff in the game.
<code>outcomes</code>	The set of outcomes in the game.
<code>players</code>	The set of players in the game.
<code>root</code>	The root node of the game.
<code>strategies</code>	The set of strategies in the game.
<code>title</code>	Get or set the title of the game.

pygambit.gambit.Player

class pygambit.gambit.Player

A player in a Game.

Methods

Attributes

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

pygambit.gambit.Outcome

class pygambit.gambit.Outcome

An outcome in a Game.

Methods

Attributes

<i>game</i>	Returns the game with which this outcome is associated.
<i>label</i>	The text label associated with this outcome.

pygambit.gambit.Node

class pygambit.gambit.Node

A node in a Game.

Methods

<code>is_successor_of(node)</code>	Returns whether this node is a successor of <i>node</i> .
------------------------------------	---

Attributes

<code>children</code>	The set of children of this node.
<code>game</code>	Gets the Game to which the node belongs.
<code>infoset</code>	The information set to which this node belongs.
<code>is_subgame_root</code>	Returns whether the node is the root of a proper subgame.
<code>is_terminal</code>	Returns whether this is a terminal node of the game.
<code>label</code>	The text label associated with the node.
<code>next_sibling</code>	The node which is immediately after this one in its parent's children.
<code>outcome</code>	Returns the outcome attached to the node.
<code>parent</code>	The parent of this node.
<code>player</code>	The player who makes the decision at this node.
<code>prior_action</code>	The action which leads to this node.
<code>prior_sibling</code>	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

<code>precedes(node)</code>	Return whether this information set precedes <i>node</i> in the game tree.
-----------------------------	--

Attributes

<code>actions</code>	The set of actions at the information set.
<code>game</code>	The Game to which the information set belongs.
<code>is_chance</code>	Whether the information set belongs to the chance player.
<code>label</code>	Get or set the text label of the information set.
<code>members</code>	The set of nodes which are members of the information set.
<code>number</code>	Returns the number of the information set for its player.
<code>player</code>	The player who has the move at this information set.

pygambit.gambit.Action**class** pygambit.gambit.Action

A choice available at an Infoset in a Game.

Methods

precedes(<i>node</i>)	Returns whether <i>node</i> 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.
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**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

<code>Game.new_tree([players, title])</code>	Create a new Game consisting of a trivial game tree, with one node, which is both root and terminal.
<code>Game.new_table(dim[, title])</code>	Create a new Game with a strategic representation.
<code>Game.from_arrays(*arrays[, title])</code>	Create a new Game with a strategic representation.
<code>Game.from_dict(payoffs[, title])</code>	Create a new Game with a strategic representation.
<code>Game.read_game(filepath)</code>	Construct a game from its serialised representation in a file.
<code>Game.parse_game(text)</code>	Construct a game from its serialised representation in a string
<code>Game.write([format])</code>	Produce a serialization of the game.

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

pygambit.gambit.Game.from_dict

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

Create a new Game with a strategic representation.

Each entry in *payloads* 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

- **payloads** (*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.read_game

classmethod `Game.read_game(filepath: str | Path) → Game`

Construct a game from its serialised representation in a file.

Parameters

filepath (*str or path object*) – The path to the file containing the game representation.

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:

[*parse_game*](#)

Constructs a game from a text string.

pygambit.gambit.Game.parse_game

classmethod `Game.parse_game(text: str) → Game`

Construct a game from its serialised representation in a string

Parameters

text (*str*) – A string containing the game representation.

Returns

A game constructed from the representation in the string.

Return type

Game

Raises

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

See also:

[*read_game*](#)

Constructs a game from a representation in a file.

pygambit.gambit.Game.write

`Game.write(format='native') → str`

Produce a serialization of the game.

Several output formats are supported, depending on the representation of the game.

- *efg*: A representation of the game in *the .efg extensive game file format*. Not available for games in strategic representation.

- *nfg*: A representation of the game in *the .nfg strategic game file format*. For an extensive game, this uses the reduced strategic form representation.
- *gte*: The XML representation used by the Game Theory Explorer tool. Only available for extensive games.
- *native*: The format most appropriate to the underlying representation of the game, i.e., *efg* or *nfg*.

This method also supports exporting to other output formats (which cannot be used directly to re-load the game later, but are suitable for human consumption, inclusion in papers, and so on).

- *html*: 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.
- *sgame*: 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.

Transforming game trees

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

pygambit.gambit.Game.append_move

`Game.append_move(nodes: Node | str | 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_infoiset

Game.**append_infoiset**(nodes: [Node](#) | *str* | Iterable[[Node](#) | *str*], infoiset: [Infoiset](#) | *str*) → None

Add a move in information set *infoiset* 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 *infoiset* is an *Infoiset* 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_infoiset

Game.**insert_infoiset**(node: [Node](#) | *str*, infoiset: [Infoiset](#) | *str*) → None

Insert a move in information set *infoiset* 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 *infoiset* is an *Infoiset* from a different game.

pygambit.gambit.Game.copy_tree

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

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

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

<code>Game.set_player</code> (infoiset, player)	Set the player at an information set.
<code>Game.set_infoiset</code> (node, infoiset)	Place <i>node</i> in the information set <i>infoiset</i> .
<code>Game.leave_infoiset</code> (node)	Remove this node from its information set.
<code>Game.reveal</code> (infoiset, player)	Reveals the move made at <i>infoiset</i> to <i>player</i> .
<code>Game.set_chance_probs</code> (infoiset, probs)	Set the action probabilities at chance information set <i>infoiset</i> .

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.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.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.

Transforming game components

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

<code>Game.title</code>	Get or set the title of the game.
<code>Game.comment</code>	Get or set the comment of the game.
<code>Game.is_const_sum</code>	Whether the game is constant sum.
<code>Game.is_tree</code>	Return whether a game has a tree-based representation.
<code>Game.is_perfect_recall</code>	Whether the game is perfect recall.
<code>Game.players</code>	The set of players in the game.
<code>Game.outcomes</code>	The set of outcomes in the game.
<code>Game.min_payoff</code>	The minimum payoff in the game.
<code>Game.max_payoff</code>	The maximum payoff in the game.
<code>Game.strategies</code>	The set of strategies in the game.
<code>Game.root</code>	The root node of the game.
<code>Game.actions</code>	The set of actions available in the game.
<code>Game.infosets</code>	The set of information sets in the game.
<code>Game.nodes([subtree])</code>	Return a list of nodes in the game tree.
<code>Game.contingencies</code>	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.comment

Game.comment

Get or set the comment of the game.

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

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 in the game.

pygambit.gambit.Game.max_payoff**Game.max_payoff**

The maximum payoff in the game.

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 have 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(subtree: Node | str | None = None) → List[Node]`

Return a list of nodes in the game tree. If *subtree* is not *None*, returns the nodes in the subtree rooted at that node.

Nodes are returned in prefix-traversal order: a node appears prior to the list of nodes in the subtrees rooted at the node's children.

Parameters

subtree ([Node](#) or *str*, *optional*) – If specified, return only the nodes in the subtree rooted at *subtree*.

Raises

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

pygambit.gambit.Game.contingencies

`Game.contingencies`

An iterator over the contingencies in the game.

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

pygambit.gambit.Player.actions**Player.actions**

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

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 outcome of the game.

pygambit.gambit.Player.max_payoff**Player.max_payoff**

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

<i>Outcome.label</i>	The text label associated with this outcome.
<i>Outcome.game</i>	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.game**Outcome.game**

Returns the game with which this outcome is associated.

<i>Node.label</i>	The text label associated with the node.
<i>Node.game</i>	Gets the Game to which the node belongs.
<i>Node.outcome</i>	Returns the outcome attached to the node.
<i>Node.children</i>	The set of children of this node.
<i>Node.parent</i>	The parent of this node.
<i>Node.is_subgame_root</i>	Returns whether the node is the root of a proper sub-game.
<i>Node.is_terminal</i>	Returns whether this is a terminal node of the game.
<i>Node.prior_action</i>	The action which leads to this node.
<i>Node.prior_sibling</i>	The node which is immediately before this one in its parent's children.
<i>Node.next_sibling</i>	The node which is immediately after this one in its parent's children.
<i>Node.infoset</i>	The information set to which this node belongs.
<i>Node.player</i>	The player who makes the decision at this node.
<i>Node.is_successor_of</i> (node)	Returns whether this node is a successor of <i>node</i> .

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.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*.

<code>Infoset.label</code>	Get or set the text label of the information set.
<code>Infoset.game</code>	The Game to which the information set belongs.
<code>Infoset.is_chance</code>	Whether the information set belongs to the chance player.
<code>Infoset.player</code>	The player who has the move at this information set.
<code>Infoset.actions</code>	The set of actions at the information set.
<code>Infoset.members</code>	The set of nodes which are members of the information set.
<code>Infoset.precedes(node)</code>	Return whether this information set precedes <i>node</i> in the game tree.

<code>Action.label</code>	Get or set the text label of the action.
<code>Action.infoset</code>	Get the information set to which the action belongs.
<code>Action.precedes(node)</code>	Returns whether <i>node</i> precedes this action in the extensive game.
<code>Action.prob</code>	Get the probability a chance action is played.

<code>Strategy.label</code>	Get or set the text label associated with the strategy.
<code>Strategy.game</code>	The game to which the strategy belongs.
<code>Strategy.player</code>	The player to which the strategy belongs.
<code>Strategy.number</code>	The number of the strategy.

Player behavior

<code>Game.mixed_strategy_profile([data, rational])</code>	Create a mixed strategy profile over the game.
<code>Game.random_strategy_profile([denom, gen])</code>	Create a <i>MixedStrategy</i> on the game, with probabilities drawn from the uniform distribution over the set of mixed strategy profiles.
<code>Game.mixed_behavior_profile([data, rational])</code>	Create a mixed behavior profile over the game.
<code>Game.random_behavior_profile([denom, gen])</code>	Create a <i>MixedBehaviorProfile</i> on the game, with probabilities drawn from the uniform distribution over the set of mixed behavior profiles.
<code>Game.support_profile()</code>	

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 = 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 = 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.support_profile`Game.support_profile()`**3.2.2 Representation of strategic behavior****Probability distributions over strategies**

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

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

Attributes

<i>game</i>	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.

New 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.

New 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.

New 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.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.

New in version 16.2.0.

See also:

[*strategy_regret*](#), [*player_regret*](#), [*liap_value*](#)

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.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**(*profile*: *MixedStrategyProfile*, *player*: *Player*)

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__()** → Iterator[Tuple[*Strategy*, ProfileDType], None, None]

Iterate over the probabilities assigned to strategies by the mixed strategy.

New 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__**(*index*: *Strategy* | *str*) → float | RationalReturns 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__(*index*: Strategy | str, *value*: Any) → None

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

<code>MixedBehaviorProfile</code>	Represents a mixed behavior profile over the actions in a <code>Game</code> .
<code>MixedBehaviorProfile.game</code>	The game on which this mixed behavior profile is defined.
<code>MixedBehaviorProfile.mixed_behaviors()</code>	Iterate over the mixed behaviors in the profile.
<code>MixedBehaviorProfile.mixed_actions()</code>	Iterate over the mixed actions specified by the profile.
<code>MixedBehaviorProfile.__iter__</code>	Iterate over the probabilities assigned to actions by the profile.
<code>MixedBehaviorProfile.__getitem__</code>	Access a component of the mixed behavior specified by <i>index</i> .
<code>MixedBehaviorProfile.__setitem__</code>	Sets a probability, mixed agent strategy, or mixed behavior strategy to <i>value</i> .
<code>MixedBehaviorProfile.payoff(player)</code>	Returns the expected payoff to a player if all players play according to the profile.
<code>MixedBehaviorProfile.action_regret(action)</code>	Returns the regret to playing <i>action</i> , if all other players play according to the profile.
<code>MixedBehaviorProfile.action_value(action)</code>	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.
<code>MixedBehaviorProfile.infoset_value(infoset)</code>	Returns the expected payoff to the player conditional on reaching an information set, if all players play according to the profile.
<code>MixedBehaviorProfile.node_value(player, node)</code>	Returns the expected payoff to <i>player</i> conditional on play reaching <i>node</i> , if all players play according to the profile.
<code>MixedBehaviorProfile.realiz_prob(node)</code>	Returns the probability with which a node is reached.
<code>MixedBehaviorProfile.infoset_prob(infoset)</code>	Returns the probability with which an information set is reached.
<code>MixedBehaviorProfile.belief(node)</code>	Returns the conditional probability that a node is reached, given that its information set is reached.
<code>MixedBehaviorProfile.is_defined_at(infoset)</code>	Returns whether the profile has probabilities defined at the information set.
<code>MixedBehaviorProfile.liap_value()</code>	Returns the Lyapunov value (see [McK91]) of the strategy profile.
<code>MixedBehaviorProfile.as_strategy()</code>	Returns a <i>MixedStrategyProfile</i> which is equivalent to the profile.
<code>MixedBehaviorProfile.normalize()</code>	Create a profile with the same action proportions as this one, but normalised so probabilities for each info set sum to one.
<code>MixedBehaviorProfile.copy()</code>	Creates a copy of the behavior strategy profile.
<code>MixedBehavior(profile, player)</code>	A set of probability distributions describing a player's behavior.
<code>MixedBehavior.mixed_actions()</code>	Iterate over the mixed actions specified by the mixed behavior.
<code>MixedBehavior.__iter__()</code>	Iterate over the probabilities assigned to actions by the mixed behavior.
<code>MixedBehavior.__getitem__(index)</code>	Access a component of the mixed behavior specified by <i>index</i> .
<code>MixedBehavior.__setitem__(index, value)</code>	Sets a component of the mixed behavior to <i>value</i> .
<code>MixedAction(profile, infoset)</code>	A probability distribution over a player's actions at an information set.
<code>MixedAction.__iter__()</code>	Iterate over the probabilities assigned to actions by the mixed action.
<code>MixedAction.__getitem__(index)</code>	Returns the probability that the action referred to by <i>index</i> is played.
<code>MixedAction.__setitem__(index, value)</code>	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, info set, 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

<code>action_regret(action)</code>	Returns the regret to playing <i>action</i> , if all other players play according to the profile.
<code>action_value(action)</code>	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.
<code>as_strategy()</code>	Returns a <i>MixedStrategyProfile</i> which is equivalent to the profile.
<code>belief(node)</code>	Returns the conditional probability that a node is reached, given that its information set is reached.
<code>copy()</code>	Creates a copy of the behavior strategy profile.
<code>infoset_prob(infoset)</code>	Returns the probability with which an information set is reached.
<code>infoset_regret(infoset)</code>	Returns the regret to the player for playing their mixed action at <i>infoset</i> , if all other players play according to the profile.
<code>infoset_value(infoset)</code>	Returns the expected payoff to the player conditional on reaching an information set, if all players play according to the profile.
<code>is_defined_at(infoset)</code>	Returns whether the profile has probabilities defined at the information set.
<code>liap_value()</code>	Returns the Lyapunov value (see [McK91]) of the strategy profile.
<code>max_regret()</code>	Returns the maximum regret of any player.
<code>mixed_actions()</code>	Iterate over the mixed actions specified by the profile.
<code>mixed_behaviors()</code>	Iterate over the mixed behaviors in the profile.
<code>node_value(player, node)</code>	Returns the expected payoff to <i>player</i> conditional on play reaching <i>node</i> , if all players play according to the profile.
<code>normalize()</code>	Create a profile with the same action proportions as this one, but normalised so probabilities for each info-set sum to one.
<code>payoff(player)</code>	Returns the expected payoff to a player if all players play according to the profile.
<code>realiz_prob(node)</code>	Returns the probability with which a node is reached.

Attributes

<code>game</code>	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.

New 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.

New 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.

New 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_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.action_value

`MixedBehaviorProfile.action_value(action: Action | str)` → float | Rational

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.

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

pygambit.gambit.MixedBehaviorProfile.infoset_value

`MixedBehaviorProfile.infoset_value(infoset: Infoset | str)` → float | Rational

Returns the expected payoff to the player conditional on reaching an information set, if all players play according to the profile.

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

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`

Returns the conditional probability that a node is reached, given that its information set is reached.

Parameters

node – The node of the game tree

Raises

MismatchError – If *node* is not in the same game as the profile

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.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 agent Nash equilibria.

See also:

`max_regret`

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 info set 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**(*profile*: *MixedBehaviorProfile*, *player*: *Player*)

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

<i>mixed_actions()</i>	Iterate over the mixed actions specified by the mixed behavior.
------------------------	---

Attributes

<i>player</i>	The player for whom this mixed behavior strategy is defined.
<i>profile</i>	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.

New in version 16.2.0.

Yields

- **info set** (*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__() → Iterator[Tuple[[Action](#), ProfileDType], None, None]

Iterate over the probabilities assigned to actions by the mixed behavior.

New 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__(*index*: [Infoset](#) | *str* | [Action](#)) → [MixedAction](#) | float | Rational

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__(*index*: [Infoset](#) | *str* | [Action](#), *value*: Any) → None

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**(*profile*: [MixedBehaviorProfile](#), *infoset*: [InfoSet](#))

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

<code>infoset</code>	The information set over which this mixed action is defined.
<code>profile</code>	The full profile of which this is a part.

pygambit.gambit.MixedAction.__iter__

MixedAction.__iter__() → Iterator[Tuple[[Action](#), ProfileDType], None, None]

Iterate over the probabilities assigned to actions by the mixed action.

New 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__(*index*: [Action](#) | *str*) → float | Rational

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__(index: Action | str, value: Any) → None`

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.

3.2.3 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: 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** (`StrategySupportProfile`) – The initial profile of strategies
- **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`

3.2.4 Computation of Nash equilibria

<code>NashComputationResult(game, method, ...)</code>	Represents the result of a method which computes Nash equilibria in a game.
<code>enumpure_solve(game[, use_strategic])</code>	Compute all <i>pure-strategy Nash equilibria</i> of game.
<code>enummixed_solve(game[, rational, use_lrs])</code>	Compute all <i>mixed-strategy Nash equilibria</i> of a two-player game using the strategic representation.
<code>lp_solve(game[, rational, use_strategic])</code>	Compute Nash equilibria of a two-player constant-sum game using <i>linear programming</i> .
<code>lcp_solve(game[, rational, use_strategic, ...])</code>	Compute Nash equilibria of a two-player game using <i>linear complementarity programming</i> .
<code>liap_solve(start[, maxregret, maxiter])</code>	Compute approximate Nash equilibria of a game using <i>Lyapunov function minimization</i> .
<code>logit_solve(game[, use_strategic, ...])</code>	Compute Nash equilibria of a game using <i>the logit quantal response equilibrium correspondence</i> .
<code>simpdiv_solve(start[, maxregret, refine, leash])</code>	Compute Nash equilibria of a game using <i>simplicial subdivision</i> .
<code>ipa_solve(perturbation)</code>	Compute Nash equilibria of a game using <i>iterated poly-matrix approximation</i> .
<code>gnm_solve(perturbation[, end_lambda, steps, ...])</code>	Compute Nash equilibria of a game using <i>a global Newton method</i> .

pygambit.nash.NashComputationResult

```
class pygambit.nash.NashComputationResult(game: ~pygambit.gambit.Game, method: str, rational: bool,
                                           use_strategic: bool, equilibria:
                                           ~typing.List[~pygambit.gambit.MixedStrategyProfile] |
                                           ~typing.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**

<i>game</i>
<i>method</i>
<i>rational</i>
<i>use_strategic</i>
<i>equilibria</i>
<i>parameters</i>

pygambit.nash.enumpure_solve

`pygambit.nash.enumpure_solve(game: Game, use_strategic: bool = True) → NashComputationResult`

Compute all *pure-strategy Nash equilibria* of game.

Parameters

- **game** (*Game*) – The game to compute equilibria in.
- **use_strategic** (*bool*, *default True*) – Whether to use the strategic form. If False, computes all agent-form pure-strategy equilibria, which consider only unilateral deviations at each individual information set.

Returns

res – The result represented as a *NashComputationResult* object.

Return type

NashComputationResult

pygambit.nash.enummixed_solve

`pygambit.nash.enummixed_solve(game: Game, rational: bool = True, use_lrs: bool = False) → 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.
- **use_lrs** (*bool*, *default False*) – If *True*, use the implementation based on `lrslib`. This is experimental.

Returns

res – The result represented as a `NashComputationResult` object.

Return type

NashComputationResult

Raises

RuntimeError – If game has more than two players.

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. If not specified, computes all accessible equilibria.
- **max_depth** (*int*, *optional*) – Maximum depth of recursion. 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.

pygambit.nash.liap_solve

`pygambit.nash.liap_solve(start: MixedStrategyProfileDouble | MixedBehaviorProfileDouble, 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.

Parameters

- **start** (*MixedStrategyProfileDouble* or *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

New in version 16.2.0.

- **first_step** (*float*, *default .03*) – The arclength of the initial step.

New in version 16.2.0.

- **max_accel** (*float*, *default 1.1*) – The maximum rate at which to lengthen the arclength step size.

New 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 that 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.

New in version 16.2.0.

- **steps** (*int*, *default* `100`) – The number of steps to take within a support cell. Larger values trade off speed for security in tracing the path.

New 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.

New in version 16.2.0.

- **local_newton_maxits** (*int*, *default 10*) – The maximum number of iterations in a local Newton method step.

New 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

3.2.5 Computation of quantal response equilibria

<code>fit_empirical(data)</code>	Use maximum likelihood estimation to estimate a quantal response equilibrium using the empirical payoff method.
<code>fit_fixedpoint(data)</code>	Use maximum likelihood estimation to find the logit quantal response equilibrium on the principal branch for a strategic game which best fits empirical frequencies of play.
<code>LogitQREMixedStrategyFitResult(data, method, ...)</code>	The result of fitting a QRE to a given probability distribution over strategies.

pygambit.qre.fit_empirical

`pygambit.qre.fit_empirical(data: MixedStrategyProfileDouble) → LogitQREMixedStrategyFitResult`

Use maximum likelihood estimation to estimate a quantal response equilibrium using the empirical payoff method. The empirical payoff method operates by ignoring the fixed-point considerations of the QRE and approximates instead by a collection of independent decision problems.¹

New in version 16.1.0.

Returns

The result of the estimation represented as a `LogitQREMixedStrategyFitResult` object.

Return type

LogitQREMixedStrategyFitResult

See also:

`fit_fixedpoint`

Estimate QRE precisely by computing the correspondence

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.

References

pygambit.qre.fit_fixedpoint

`pygambit.qre.fit_fixedpoint(data: MixedStrategyProfileDouble) → LogitQREMixedStrategyFitResult`

Use maximum likelihood estimation to find the logit quantal response equilibrium on the principal branch for a strategic game which best fits empirical frequencies of play.¹

New in version 16.1.0.

Parameters

data (*MixedStrategyProfileDouble*) – 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 strategy rather than probabilities.

Returns

The result of the estimation represented as a `LogitQREMixedStrategyFitResult` object.

Return type

LogitQREMixedStrategyFitResult

See also:

[*fit_empirical*](#)

Estimate QRE by approximation of the correspondence using independent decision problems.

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.

See also:

[*fit_fixedpoint*](#), [*fit_empirical*](#)

Methods

Attributes

<code>data</code>	The empirical strategy frequencies used to estimate the QRE.
<code>lam</code>	The value of lambda corresponding to the QRE.
<code>log_like</code>	The log-likelihood of the data at the estimated QRE.
<code>method</code>	The method used to estimate the QRE; either "fixed-point" or "empirical".
<code>profile</code>	The mixed strategy profile corresponding to the QRE.

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.

THE GRAPHICAL INTERFACE

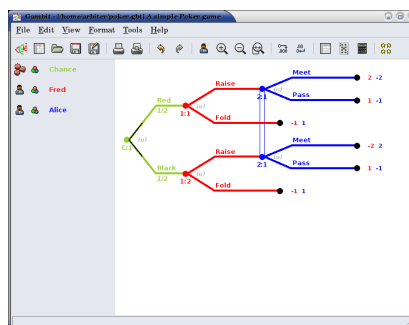
Gambit’s graphical user interface provides an “integrated development environment” to help visually construct games and to investigate their main strategic features.

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*.

4.1 General concepts

4.1.1 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 and actions*.

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](#).

4.1.2 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.

4.1.3 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.

4.2 Extensive games

The graphical interface provides a flexible set of operations for constructing and editing general extensive games. These are outlined below.

4.2.1 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.

4.2.2 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.

4.2.3 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.

4.2.4 Removing parts of a game tree

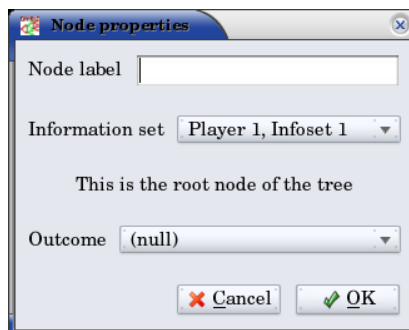
Two deletion operations are supported on extensive games. To delete the entire subtree rooted at a node, click on that node and select *Edit* → *Delete subtree*.

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.

4.2.5 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.

4.2.6 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.

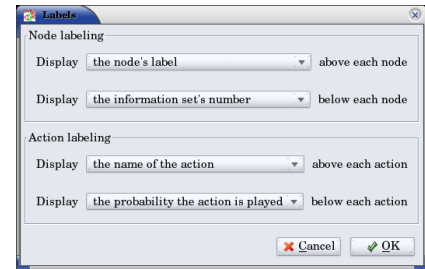
To remove an outcome from a node, click on the node, and select *Edit → Remove outcome*.

4.2.7 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 is the default labeling above the branch.)

The probability the action is played

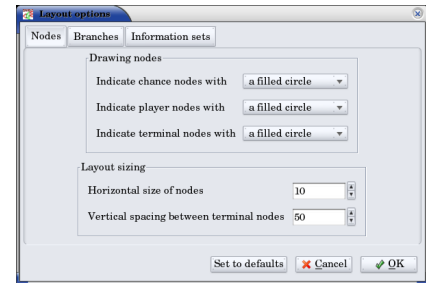
For chance actions, the probability the branch is taken; this is always displayed. For player actions, the probability the action is taken in the selected profile (only displayed when a behavior strategy is selected to be displayed on the tree). In some cases, behavior strategies do not fully specify 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.)

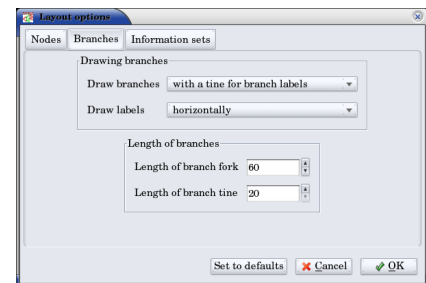
Controlling the layout of the tree

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.

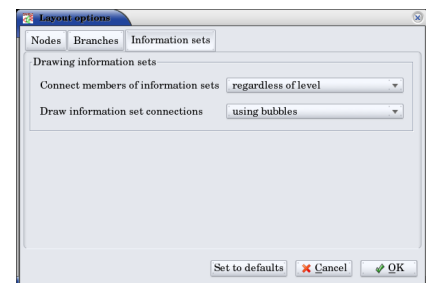
The sizing of nodes can be configured for best results. Gambit styling from previous versions used the horizontal line tokens with relatively long lines; when using the other tokens, smaller node sizes often look better.



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.

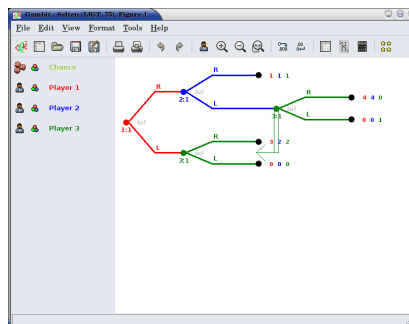
The rotated label drawing is experimental, and does not always look good on screen.



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.

4.3 Strategic games

Gambit has full support for constructing and manipulating arbitrary N-player strategic (also known as normal form) games.

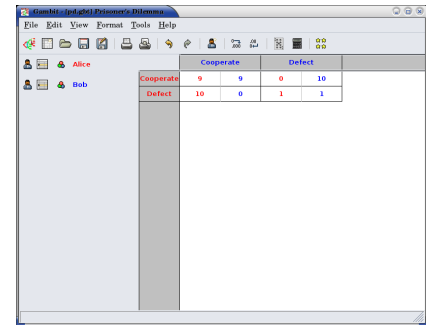
For extensive games, Gambit automatically computes the corresponding reduced strategic game. To view the reduced strategic game corresponding to an extensive game, select *View* → *Strategic game* or click the strategic game table icon on the toolbar.

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.

4.3.1 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.

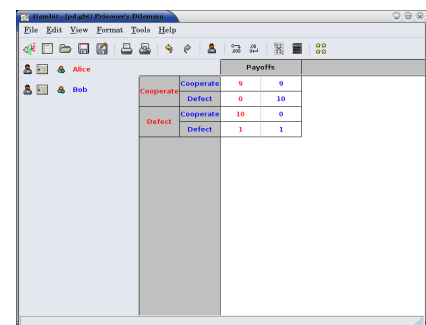


		Bob	
		Cooperate	Defect
Alice	Cooperate	9, 9	0, 10
	Defect	10, 0	1, 1

For games with two players, this presentation is by default configured to be similar to the standard presentation 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.



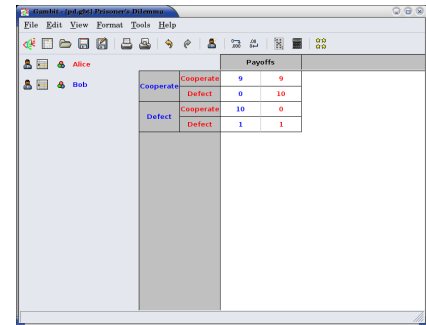
		Payoffs	
		Bob	Alice
Alice	Cooperate	9, 9	0, 10
	Defect	10, 0	1, 1

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.



		Payoffs	
Alice	Cooperate	9	9
	Defect	0	10
Bob	Cooperate	9	0
	Defect	1	1

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.

4.3.2 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.

4.3.3 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.

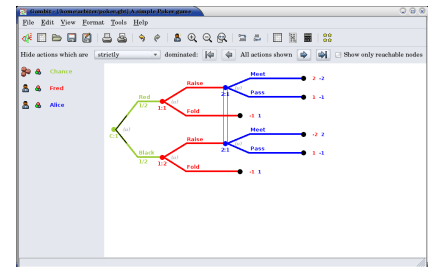
To speed entry of many payoffs, as is typical when creating a new game, accepting a payoff entry via the Tab key automatically moves the edit control to the next cell to the right. If the payoff is the last payoff listed in a row of the table, the edit control wraps around to the first payoff in the 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.

4.4 Investigating dominated strategies and actions

Selecting *Tools* → *Dominance* toggles the appearance of a toolbar which can be used to investigate the structure of dominated strategies and actions.

4.4.1 Dominated actions in extensive game

In extensive games, the dominance toolbar controls the elimination of actions which are conditionally dominated.



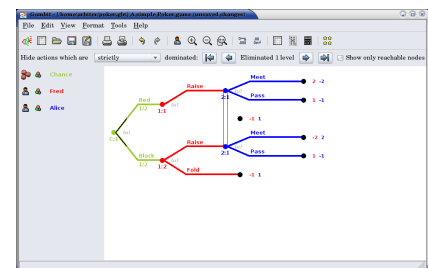
Actions may be eliminated based on two criteria:

Strict dominance

The action is always worse than another, regardless of beliefs at the information set;

Strict or weak dominance

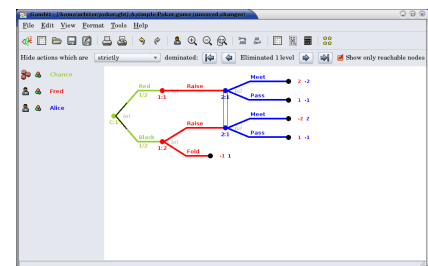
There is another action at the information set that is always at least as good as the action, and strictly better in some cases.



For example, in the poker game, it is strictly dominated for Fred to choose Fold after Red. Clicking the next level icon removes the dominated action from the game display.

The tree layout remains unchanged, including nodes which can only be reached using actions which have been eliminated. To compress the tree to remove the unreachable nodes, check the box labeled *Show only reachable nodes*.

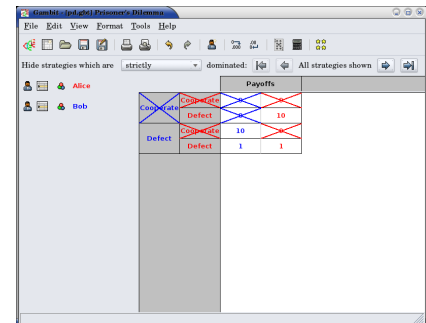
For this game, no further actions can be eliminated. In general, further steps of elimination can be done by again clicking the next level icon. The toolbar keeps track of the number of levels of elimination currently shown; the previous level icon moves up one level of elimination.



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.

4.4.2 Dominated strategies in strategic games

The dominance toolbar operates in strategic games in the same way as the in the extensive game. Strategies can be eliminated iteratively based on whether they are strictly or weakly dominated.

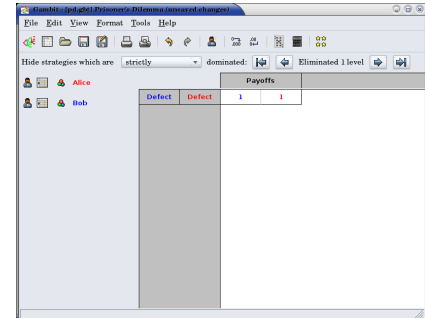


The screenshot shows the Gambit interface for a Prisoner's Dilemma game. The payoff matrix is displayed with Alice's strategies (Cooperate, Defect) on the rows and Bob's strategies (Cooperate, Defect) on the columns. The payoffs are (10, 10) for (Cooperate, Cooperate), (1, 10) for (Cooperate, Defect), (10, 1) for (Defect, Cooperate), and (1, 1) for (Defect, Defect). Thick solid 'X' marks are drawn over the 'Cooperate' strategy labels for both players and the corresponding payoff cells, indicating strict dominance. A toolbar at the top includes icons for dominance analysis, and a dropdown menu shows 'Hide strategies which are' set to 'strictly'.

		Bob		Payoffs
		Cooperate	Defect	
Alice	Cooperate	10, 10	1, 10	
	Defect	10, 1	1, 1	

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.



This screenshot shows the same Gambit interface after clicking the 'next level' icon. The strictly dominated 'Cooperate' strategies have been removed from the display. The payoff matrix now only shows the (Defect, Defect) outcome with a payoff of (1, 1). The toolbar dropdown menu now shows 'Eliminated 1 level'.

		Bob		Payoffs
		Cooperate	Defect	
Alice	Defect		1, 1	

Clicking the next level icon removes the strictly dominated strategies from the display.

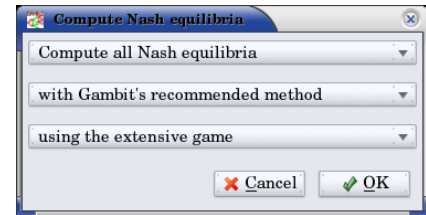
4.5 Computing Nash equilibria

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.

4.5.1 Selecting the method of computing equilibria

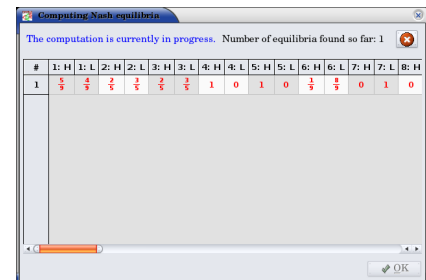
The process of computing Nash equilibria in extensive and strategic games is similar. This section focuses on the case of extensive games; the process for strategic games is analogous, except the extensive game-specific features, such as displaying the profiles on the game tree, are not applicable.

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 [Command-line tools](#).

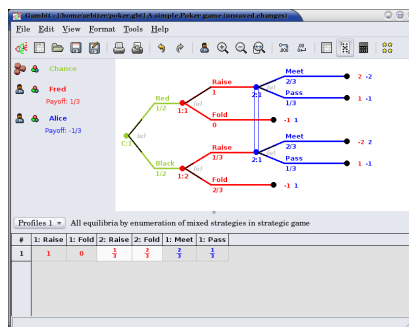


Finally, for extensive games, there is an option of whether to use the extensive or strategic game for computation. In general, computation using the extensive game is preferred, since it is often a significantly more compact representation of the strategic characteristics of the game than the reduced 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.

4.5.2 Viewing computed profiles in the game

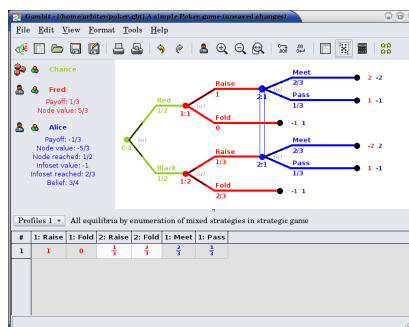
After computing equilibria, a panel showing the list of equilibria computed is displayed automatically. The display of this panel can be toggled by selecting *View* → *Profiles*, or clicking on the playing card icon on the toolbar.



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.

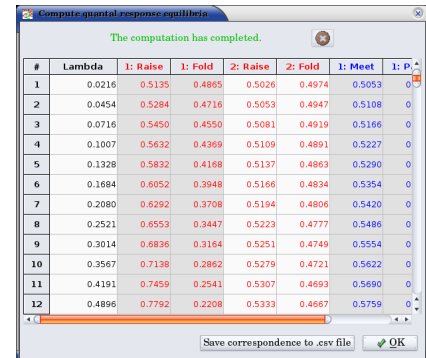


Clicking on any node in the tree displays additional information about the profile at that node. The player panel displays information relevant to the selected node, including the payoff to all players conditional on reaching the node, as well as information about Alice's beliefs at the node.

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.

4.5.3 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].



The computation has completed.

#	Lambda	1: Raise	1: Fold	2: Raise	2: Fold	1: Meet	1: P
1	0.0216	0.5135	0.4865	0.5026	0.4974	0.5053	0
2	0.0454	0.5284	0.4716	0.5053	0.4947	0.5108	0
3	0.0716	0.5450	0.4550	0.5081	0.4919	0.5166	0
4	0.1007	0.5632	0.4368	0.5109	0.4891	0.5227	0
5	0.1328	0.5832	0.4168	0.5137	0.4863	0.5290	0
6	0.1684	0.6052	0.3948	0.5166	0.4834	0.5354	0
7	0.2080	0.6292	0.3708	0.5194	0.4806	0.5420	0
8	0.2521	0.6553	0.3447	0.5223	0.4777	0.5486	0
9	0.3014	0.6836	0.3164	0.5251	0.4749	0.5554	0
10	0.3567	0.7138	0.2862	0.5279	0.4721	0.5622	0
11	0.4191	0.7459	0.2541	0.5307	0.4693	0.5690	0
12	0.4896	0.7792	0.2208	0.5333	0.4667	0.5758	0

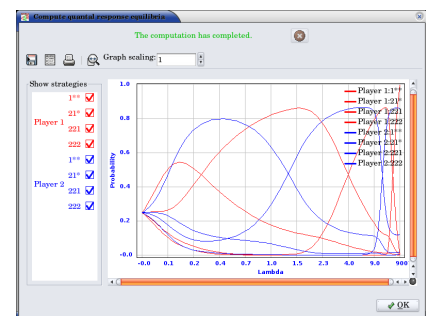
Save correspondence to .csv file OK

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.

4.5.4 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.

4.6 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.

4.6.1 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.

4.6.2 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.

SAMPLE GAMES

2x2x2.nfg

A three-player normal form game with two strategies per player. This game has nine Nash equilibria, which is the maximal number of regular Nash equilibria possible for a game of this size. See McKelvey, Richard D. and McLennan, Andrew (1997). The maximal number of regular totally mixed Nash equilibria. *Journal of Economic Theory* 72(2): 411-425.

2x2x2-nau.nfg

A three-player normal form game with two strategies per player. This game has three pure strategy equilibria, two equilibria which are incompletely mixed, and a continuum of completely mixed equilibria. This game appears as an example in Nau, Robert, Gomez Canovas, Sabrina, and Hansen, Pierre (2004). On the geometry of Nash equilibria and correlated equilibria. *International Journal of Game Theory* 32(4): 443-453.

bagwell.efg

Stackelberg leader game with imperfectly observed commitment, from Bagwell, Kyle (1993) Commitment and observability in games. *Games and Economic Behavior* 8: 271-280.

bayes2a.efg

A twice-repeated Bayesian game, with two players, each having two types and two actions. This game also illustrates the use of payoffs at nonterminal nodes in Gambit, which can substantially simplify the representation of multi-stage games such as this.

cent3.efg

A three-stage centipede game, featuring an exogenous probability that one player is an altruistic type, who always passes. See, for example, McKelvey, Richard D. and Palfrey, Thomas R. (1992) An experimental study of the centipede game. *Econometrica* 60(4): 803-836.

condjury.efg

A three-person Condorcet jury game, after the analysis of Feddersen, Timothy and Pesendorfer, Wolfgang (1998) Convicting the innocent: The inferiority of unanimous jury verdicts under strategic voting. *American Political Science Review* 92(1): 23-35..

loopback.nfg

A game due to McKelvey which illustrates that the logit quantal response equilibrium correspondence can have a “backward-bending” segment on the principal branch.

montyhal.efg

The famous [Monty Hall problem](#): if Monty offers to let you switch doors, should you?

nim.efg

The classic game of [Nim](#), which is a useful example of the value of backward induction. This version starts with five stones. An interesting experimental study of this class of games is McKinney, C. Nicholas and Van Huyck, John B. (2013) Eureka learning: Heuristics and response time in perfect information games. *Games and Economic Behavior* 79: 223-232.

pbride.efg

A signaling game from [Joel Watson's Strategy textbook](#), modeling the confrontation in The Princess Bride between Humperdinck and Roberts in the bedchamber.

poker.efg

A simple game of one-card poker introduced in [Myerson, Roger \(1991\) Game Theory: Analysis of Conflict](#). A bit unusually for poker, the “fold” action by a player with a strong hand counts for a win for that player, so folding is only weakly rather than strictly dominated in this case.

4cards.efg

A slightly more complex poker example, contributed by Alix Martin.

spence.efg

A version of Spence's classic job-market signaling game. This version comes from [Joel Watson's Strategy textbook](#).

These games, and others, ship in the standard Gambit source distribution in the directory *contrib/games*.

FOR DEVELOPERS: BUILDING GAMBIT FROM SOURCE

This section 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 section requires at least some familiarity with programming. Most users will want to stick with binary distributions; see [Downloading Gambit](#) for how to get the current version for your operating system.

6.1 General information

Gambit uses the standard autotools mechanism for configuring and building. This should be familiar to most users of Un*xes and MacOS X.

If you are building from a source tarball, you just need to unpack the sources, change directory to the top level of the sources (typically of the form `gambit-xx.y.z`), and do the usual

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

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: 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!

6.2 Building from git repository

If you want to live on the bleeding edge, you can get the latest version of the Gambit sources from the Gambit repository on github.com, via

```
git clone https://github.com/gambitproject/gambit.git
cd gambit
```

After this, you will need to set up the build scripts by executing

```
aclocal
libtoolize
automake --add-missing
autoconf
```

For this, you will need to have automake, autoconf, and libtool2 installed on your system.

At this point, you can then continue with the configuration and build stages as in the previous section.

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 `maintVV`, where `VV` is the version number, point to the latest commit on a stable version.

6.3 For Windows users

For Windows users wanting to compile Gambit on their own, you'll need to use either the Cygwin or MinGW environments. We do compilation and testing of Gambit on Windows using MinGW.

6.4 For OS X users

For building the command-line tools only, one should follow the instructions for Un*x/Linux platforms above.

6.5 The graphical interface and wxWidgets

Gambit requires wxWidgets version 3.1.x or higher. See the wxWidgets website at <http://www.wxwidgets.org> to download this if you need it. Packages of this should be available for most Un*x users through their package managers (apt or rpm). Note that you'll need the appropriate `-dev` package for wxWidgets to get the header files needed to build Gambit.

Un*x users, please note that Gambit at this time only supports the GTK port of wxWidgets.

If wxWidgets it 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
```

Finally, if you don't want to build the graphical interface, you can either (a) simply not install wxWidgets, or (b) pass the argument `--disable-gui` to the configure step, for example,

```
./configure --disable-gui
```

This will just build the command-line tools, and will not require a wxWidgets installation.

For OS X users, after the usual `make` step, run

```
make osx-bundle
```

This produces an application `Gambit.app` in the current directory, which can be run from its current location, or copied elsewhere in the disk (such as `/Applications`). The application bundle includes the command-line executables.

6.6 Building the Python extension

The *pygambit Python package* is in `src/pygambit` in the Gambit source tree. Building the extension follows the standard approach. From the **root directory of the source tree** execute

```
python -m pip install .
```

There is a set of test cases in `src/pygambit/tests`, which can be run using *nose2*.

Once installed, simply `import pygambit` in your Python shell or script to get started.

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.

7.1 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 formatings 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 internally 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.

7.1.1 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. It is intended that some new formats will be introduced in the future; however, this format will be supported by Gambit, possibly through the use of converter programs to those putative future formats, for the foreseeable future.

A sample file

This is a sample file illustrating the general format of the file. This file is similar to the one distributed in the Gambit distribution under the name bayes1a.efg:

```
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 indicates the null outcome.

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. 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 omissible 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
- (optional) a list of actions at the information set with their corresponding probabilities
- a nonnegative integer specifying the outcome
- (optional) 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
- (optional) a list of action names for the information set
- a nonnegative integer specifying the outcome
- (optional) the name of the outcome
- 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
- the payoffs to each player for the outcome

There is no explicit end-of-file delimiter for the file.

7.1.2 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.

7.1.3 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.

7.1.4 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:
 $|S_1| |S_2| \dots |S_n|$
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 Δ_s , the number of elements in the mapping. Then follows Δ_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
```

7.1.5 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`**.

BIBLIOGRAPHY

8.1 Articles on computation of Nash equilibria

8.2 General game theory articles and texts

8.3 Textbooks and general reference

DETAILED TABLE OF CONTENTS

BIBLIOGRAPHY

- [Kre90] Kreps, D. (1990) “Corporate Culture and Economic Theory.” In J. Alt and K. Shepsle, eds., *Perspectives on Positive Political Economy*, Cambridge University Press.
- [Mye91] Myerson, Roger B. (1991) *Game Theory: Analysis of Conflict*. Cambridge: Harvard University Press.
- [RUW08] 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.
- [BlaTur23] Bland, J. R. and Turocy, T. L., 2023. Quantal response equilibrium as a structural model for estimation: The missing manual. SSRN working paper 4425515.
- [Eav71] B. C. Eaves, “The linear complementarity problem”, 612-634, *Management Science* , 17, 1971.
- [GovWil03] Govindan, Srihari and Robert Wilson. (2003) “A Global Newton Method to Compute Nash Equilibria.” *Journal of Economic Theory* 110(1): 65-86.
- [GovWil04] Govindan, Srihari and Robert Wilson. (2004) “Computing Nash Equilibria by Iterated Polymatrix Approximation.” *Journal of Economic Dynamics and Control* 28: 1229-1241.
- [Jiang11] A. X. Jiang, K. Leyton-Brown, and N. Bhat. (2011) “Action-Graph Games.” *Games and Economic Behavior* 71(1): 141-173.
- [KolMegSte94] Daphne Koller, Nimrod Megiddo, and Bernhard von Stengel (1996). “Efficient computation of equilibria for extensive two-person games.” *Games and Economic Behavior* 14: 247-259.
- [LemHow64] C. E. Lemke and J. T. Howson, “Equilibrium points of bimatrix games”, 413-423, *Journal of the Society of Industrial and Applied Mathematics* , 12, 1964.
- [Man64] O. Mangasarian, “Equilibrium points in bimatrix games”, 778-780, *Journal of the Society for Industrial and Applied Mathematics*, 12, 1964.
- [McK91] Richard McKelvey, A Liapunov function for Nash equilibria, 1991, California Institute of Technology.
- [McKMcl96] Richard McKelvey and Andrew McLennan, “Computation of equilibria in finite games”, 87-142, *Handbook of Computational Economics* , Edited by H. Amman, D. Kendrick, J. Rust, Elsevier, 1996.
- [PNS04] Ryan Porter, Eugene Nudelman, and Yoav Shoham. “Simple search methods for finding a Nash equilibrium.” *Games and Economic Behavior* 664-669, 2004.
- [Ros71] J. Rosenmuller, “On a generalization of the Lemke-Howson Algorithm to noncooperative n-person games”, 73-79, *SIAM Journal of Applied Mathematics*, 21, 1971.
- [Sha74] Lloyd Shapley, “A note on the Lemke-Howson algorithm”, 175-189, *Mathematical Programming Study* , 1, 1974.
- [Tur05] Theodore L. Turocy, “A dynamic homotopy interpretation of the logistic quantal response equilibrium correspondence”, 243-263, *Games and Economic Behavior*, 51, 2005.

- [Tur10] Theodore L. Turocy, “Using Quantal Response to Compute Nash and Sequential Equilibria.” *Economic Theory* 42(1): 255-269, 2010.
- [VTH87] G. van der Laan, A. J. J. Talman, and L. van Der Heyden, “Simplicial variable dimension algorithms for solving the nonlinear complementarity problem on a product of unit simplices using a general labelling”, 377-397, *Mathematics of Operations Research* , 1987.
- [Wil71] Robert Wilson, “Computing equilibria of n-person games”, 80-87, *SIAM Applied Math*, 21, 1971.
- [Yam93] Y. Yamamoto, 1993, “A Path-Following Procedure to Find a Proper Equilibrium of Finite Games “, *International Journal of Game Theory* .
- [Harsanyi1967a] John Harsanyi, “Games of Incomplete Information Played By Bayesian Players I”, 159-182, *Management Science* , 14, 1967.
- [Harsanyi1967b] John Harsanyi, “Games of Incomplete Information Played By Bayesian Players II”, 320-334, *Management Science* , 14, 1967.
- [Harsanyi1968] John Harsanyi, “Games of Incomplete Information Played By Bayesian Players III”, 486-502, *Management Science* , 14, 1968.
- [KreWil82] David Kreps and Robert Wilson, “Sequential Equilibria”, 863-894, *Econometrica* , 50, 1982.
- [McKPal95] Richard McKelvey and Tom Palfrey, “Quantal response equilibria for normal form games”, 6-38, *Games and Economic Behavior* , 10, 1995.
- [McKPal98] Richard McKelvey and Tom Palfrey, “Quantal response equilibria for extensive form games”, 9-41, *Experimental Economics* , 1, 1998.
- [Mye78] Roger Myerson, “Refinements of the Nash equilibrium concept”, 73-80, *International Journal of Game Theory* , 7, 1978.
- [Nas50] John Nash, “Equilibrium points in n-person games”, 48-49, *Proceedings of the National Academy of Sciences* , 36, 1950.
- [Och95] Jack Ochs, “Games with unique, mixed strategy equilibria: An experimental study”, *Games and Economic Behavior* 10: 202-217, 1995.
- [Sel75] Reinhard Selten, Reexamination of the perfectness concept for equilibrium points in extensive games , 25-55, *International Journal of Game Theory* , 4, 1975.
- [vanD83] Eric van Damme, 1983, *Stability and Perfection of Nash Equilibria* , Springer-Verlag, Berlin.
- [Mye91] Roger Myerson, 1991, *Game Theory : Analysis of Conflict* , Harvard University Press.

Symbols

- `__getitem__()` (*pygambit.gambit.MixedAction method*), 77
- `__getitem__()` (*pygambit.gambit.MixedBehavior method*), 76
- `__getitem__()` (*pygambit.gambit.MixedBehaviorProfile method*), 70
- `__getitem__()` (*pygambit.gambit.MixedStrategy method*), 65
- `__getitem__()` (*pygambit.gambit.MixedStrategyProfile method*), 61
- `__iter__()` (*pygambit.gambit.MixedAction method*), 77
- `__iter__()` (*pygambit.gambit.MixedBehavior method*), 76
- `__iter__()` (*pygambit.gambit.MixedBehaviorProfile method*), 70
- `__iter__()` (*pygambit.gambit.MixedStrategy method*), 65
- `__iter__()` (*pygambit.gambit.MixedStrategyProfile method*), 61
- `__setitem__()` (*pygambit.gambit.MixedAction method*), 78
- `__setitem__()` (*pygambit.gambit.MixedBehavior method*), 76
- `__setitem__()` (*pygambit.gambit.MixedBehaviorProfile method*), 71
- `__setitem__()` (*pygambit.gambit.MixedStrategy method*), 66
- `__setitem__()` (*pygambit.gambit.MixedStrategyProfile method*), 62
- A
 - gambit-enumpure command line option, 7
- D
 - gambit-enummixed command line option, 9
 - gambit-enumpure command line option, 7
 - gambit-lcp command line option, 10
 - gambit-lp command line option, 11
- L
 - gambit-enummixed command line option, 9
- O
 - gambit-convert command line option, 18
- P
 - gambit-enumpure command line option, 7
 - gambit-lcp command line option, 10
 - gambit-lp command line option, 11
- S
 - gambit-enumpure command line option, 7
 - gambit-lcp command line option, 10
 - gambit-liap command line option, 12
 - gambit-logit command line option, 15
 - gambit-lp command line option, 11
- a
 - gambit-logit command line option, 15
- C
 - gambit-convert command line option, 18
 - gambit-enummixed command line option, 9
 - gambit-gnm command line option, 16
- d
 - gambit-enummixed command line option, 8
 - gambit-gnm command line option, 16
 - gambit-ipa command line option, 17
 - gambit-lcp command line option, 10
 - gambit-liap command line option, 12
 - gambit-logit command line option, 14
 - gambit-lp command line option, 11
 - gambit-simpdiv command line option, 14
- e
 - gambit-logit command line option, 15
- f
 - gambit-gnm command line option, 16
- g
 - gambit-simpdiv command line option, 13
- h
 - gambit-convert command line option, 18
 - gambit-enummixed command line option, 9
 - gambit-enumpure command line option, 8
 - gambit-gnm command line option, 16
 - gambit-ipa command line option, 17
 - gambit-lcp command line option, 10
 - gambit-liap command line option, 12
 - gambit-logit command line option, 15
 - gambit-lp command line option, 11

gambit-simpdiv command line option, 13
 -i
 gambit-gnm command line option, 16
 gambit-liap command line option, 12
 -l
 gambit-logit command line option, 15
 -m
 gambit-gnm command line option, 16
 gambit-liap command line option, 12
 gambit-logit command line option, 15
 gambit-simpdiv command line option, 13
 -n
 gambit-gnm command line option, 16
 gambit-ipa command line option, 17
 gambit-liap command line option, 12
 gambit-simpdiv command line option, 13
 -q
 gambit-convert command line option, 18
 gambit-enummixed command line option, 9
 gambit-enumpure command line option, 8
 gambit-gnm command line option, 16
 gambit-ipa command line option, 17
 gambit-lcp command line option, 10
 gambit-liap command line option, 12
 gambit-lp command line option, 11
 gambit-simpdiv command line option, 13
 -r
 gambit-convert command line option, 18
 gambit-simpdiv command line option, 13
 -s
 gambit-gnm command line option, 16
 gambit-ipa command line option, 17
 gambit-liap command line option, 12
 gambit-logit command line option, 15
 gambit-simpdiv command line option, 13
 -v
 gambit-gnm command line option, 17
 gambit-liap command line option, 12
 gambit-simpdiv command line option, 14

A

Action (*class in pygambit.gambit*), 39

action_regret() (*pygambit.gambit.MixedBehaviorProfile method*), 71

action_value() (*pygambit.gambit.MixedBehaviorProfile method*), 72

actions (*pygambit.gambit.Game attribute*), 51

actions (*pygambit.gambit.Player attribute*), 53

add_outcome() (*pygambit.gambit.Game method*), 48

add_player() (*pygambit.gambit.Game method*), 47

add_strategy() (*pygambit.gambit.Game method*), 49

append_info() (*pygambit.gambit.Game method*), 44

append_move() (*pygambit.gambit.Game method*), 43

as_behavior() (*pygambit.gambit.MixedStrategyProfile method*), 64

as_strategy() (*pygambit.gambit.MixedBehaviorProfile method*), 74

B

belief() (*pygambit.gambit.MixedBehaviorProfile method*), 74

C

children (*pygambit.gambit.Node attribute*), 55

comment (*pygambit.gambit.Game attribute*), 50

contingencies (*pygambit.gambit.Game attribute*), 52

copy() (*pygambit.gambit.MixedBehaviorProfile method*), 75

copy() (*pygambit.gambit.MixedStrategyProfile method*), 65

copy_tree() (*pygambit.gambit.Game method*), 44

D

delete_outcome() (*pygambit.gambit.Game method*), 48

delete_parent() (*pygambit.gambit.Game method*), 45

delete_strategy() (*pygambit.gambit.Game method*), 49

delete_tree() (*pygambit.gambit.Game method*), 45

E

enummixed_solve() (*in module pygambit.nash*), 81

enumpure_solve() (*in module pygambit.nash*), 80

equilibria (*pygambit.nash.NashComputationResult attribute*), 79

F

fit_empirical() (*in module pygambit.qre*), 85

fit_fixedpoint() (*in module pygambit.qre*), 86

from_arrays() (*pygambit.gambit.Game class method*), 41

from_dict() (*pygambit.gambit.Game class method*), 41

G

gambit-convert command line option

-0, 18

-c, 18

-h, 18

-q, 18

-r, 18

gambit-enummixed command line option

-D, 9

-L, 9
 -c, 9
 -d, 8
 -h, 9
 -q, 9
 gambit-enumpure command line option
 -A, 7
 -D, 7
 -P, 7
 -S, 7
 -h, 8
 -q, 8
 gambit-gnm command line option
 -c, 16
 -d, 16
 -f, 16
 -h, 16
 -i, 16
 -m, 16
 -n, 16
 -q, 16
 -s, 16
 -v, 17
 gambit-ipa command line option
 -d, 17
 -h, 17
 -n, 17
 -q, 17
 -s, 17
 gambit-lcp command line option
 -D, 10
 -P, 10
 -S, 10
 -d, 10
 -h, 10
 -q, 10
 gambit-liap command line option
 -S, 12
 -d, 12
 -h, 12
 -i, 12
 -m, 12
 -n, 12
 -q, 12
 -s, 12
 -v, 12
 gambit-logit command line option
 -S, 15
 -a, 15
 -d, 14
 -e, 15
 -h, 15
 -l, 15
 -m, 15
 -s, 15
 gambit-lp command line option
 -D, 11
 -P, 11
 -S, 11
 -d, 11
 -h, 11
 -q, 11
 gambit-simpdiv command line option
 -d, 14
 -g, 13
 -h, 13
 -m, 13
 -n, 13
 -q, 13
 -r, 13
 -s, 13
 -v, 14
 Game (*class in pygambit.gambit*), 35
 game (*pygambit.gambit.MixedBehaviorProfile attribute*), 70
 game (*pygambit.gambit.MixedStrategyProfile attribute*), 61
 game (*pygambit.gambit.Node attribute*), 54
 game (*pygambit.gambit.Outcome attribute*), 54
 game (*pygambit.gambit.Player attribute*), 53
 game (*pygambit.nash.NashComputationResult attribute*), 79
 gnm_solve() (*in module pygambit.nash*), 84
 |
 Infoset (*class in pygambit.gambit*), 38
 infoset (*pygambit.gambit.Node attribute*), 56
 infoset_prob() (*pygambit.gambit.MixedBehaviorProfile method*), 73
 infoset_value() (*pygambit.gambit.MixedBehaviorProfile method*), 72
 infosets (*pygambit.gambit.Game attribute*), 51
 infosets (*pygambit.gambit.Player attribute*), 53
 insert_infoset() (*pygambit.gambit.Game method*), 44
 insert_move() (*pygambit.gambit.Game method*), 44
 ipa_solve() (*in module pygambit.nash*), 84
 is_chance (*pygambit.gambit.Player attribute*), 53
 is_const_sum (*pygambit.gambit.Game attribute*), 50
 is_defined_at() (*pygambit.gambit.MixedBehaviorProfile method*), 74
 is_perfect_recall (*pygambit.gambit.Game attribute*), 50
 is_subgame_root (*pygambit.gambit.Node attribute*), 55

`is_successor_of()` (*pygambit.gambit.Node* method), 56

`is_terminal` (*pygambit.gambit.Node* attribute), 55

`is_tree` (*pygambit.gambit.Game* attribute), 50

L

`label` (*pygambit.gambit.Node* attribute), 54

`label` (*pygambit.gambit.Outcome* attribute), 54

`label` (*pygambit.gambit.Player* attribute), 52

`lcp_solve()` (in module *pygambit.nash*), 82

`leave_infoaset()` (*pygambit.gambit.Game* method), 46

`liap_solve()` (in module *pygambit.nash*), 82

`liap_value()` (*pygambit.gambit.MixedBehaviorProfile* method), 74

`liap_value()` (*pygambit.gambit.MixedStrategyProfile* method), 64

`logit_solve()` (in module *pygambit.nash*), 83

`LogitQREMixedStrategyFitResult` (class in *pygambit.qre*), 86

`lp_solve()` (in module *pygambit.nash*), 81

M

`max_payoff` (*pygambit.gambit.Game* attribute), 51

`max_payoff` (*pygambit.gambit.Player* attribute), 53

`max_regret()` (*pygambit.gambit.MixedStrategyProfile* method), 63

`method` (*pygambit.nash.NashComputationResult* attribute), 79

`min_payoff` (*pygambit.gambit.Game* attribute), 51

`min_payoff` (*pygambit.gambit.Player* attribute), 53

`mixed_actions()` (*pygambit.gambit.MixedBehavior* method), 75

`mixed_actions()` (*pygambit.gambit.MixedBehaviorProfile* method), 70

`mixed_behavior_profile()` (*pygambit.gambit.Game* method), 58

`mixed_behaviors()` (*pygambit.gambit.MixedBehaviorProfile* method), 70

`mixed_strategies()` (*pygambit.gambit.MixedStrategyProfile* method), 61

`mixed_strategy_profile()` (*pygambit.gambit.Game* method), 57

`MixedAction` (class in *pygambit.gambit*), 77

`MixedBehavior` (class in *pygambit.gambit*), 75

`MixedBehaviorProfile` (class in *pygambit.gambit*), 68

`MixedStrategy` (class in *pygambit.gambit*), 65

`MixedStrategyProfile` (class in *pygambit.gambit*), 60

`move_tree()` (*pygambit.gambit.Game* method), 45

N

`NashComputationResult` (class in *pygambit.nash*), 79

`new_table()` (*pygambit.gambit.Game* class method), 40

`new_tree()` (*pygambit.gambit.Game* class method), 40

`next_sibling` (*pygambit.gambit.Node* attribute), 55

`Node` (class in *pygambit.gambit*), 37

`node_value()` (*pygambit.gambit.MixedBehaviorProfile* method), 73

`nodes()` (*pygambit.gambit.Game* method), 52

`normalize()` (*pygambit.gambit.MixedBehaviorProfile* method), 75

`normalize()` (*pygambit.gambit.MixedStrategyProfile* method), 64

`number` (*pygambit.gambit.Player* attribute), 52

O

`Outcome` (class in *pygambit.gambit*), 37

`outcome` (*pygambit.gambit.Node* attribute), 54

`outcomes` (*pygambit.gambit.Game* attribute), 50

P

`parameters` (*pygambit.nash.NashComputationResult* attribute), 80

`parent` (*pygambit.gambit.Node* attribute), 55

`parse_game()` (*pygambit.gambit.Game* class method), 42

`payoff()` (*pygambit.gambit.MixedBehaviorProfile* method), 71

`payoff()` (*pygambit.gambit.MixedStrategyProfile* method), 62

`Player` (class in *pygambit.gambit*), 36

`player` (*pygambit.gambit.Node* attribute), 56

`player_regret()` (*pygambit.gambit.MixedStrategyProfile* method), 63

`players` (*pygambit.gambit.Game* attribute), 50

`prior_action` (*pygambit.gambit.Node* attribute), 55

`prior_sibling` (*pygambit.gambit.Node* attribute), 55

R

`random_behavior_profile()` (*pygambit.gambit.Game* method), 58

`random_strategy_profile()` (*pygambit.gambit.Game* method), 57

`rational` (*pygambit.nash.NashComputationResult* attribute), 79

`read_game()` (*pygambit.gambit.Game* class method), 42

`realiz_prob()` (*pygambit.gambit.MixedBehaviorProfile* method), 73

`reveal()` (*pygambit.gambit.Game* method), 46

`root` (*pygambit.gambit.Game* attribute), 51

S

`set_chance_probs()` (*pygambit.gambit.Game* method), 47

[set_infoset\(\)](#) (*pygambit.gambit.Game method*), [46](#)
[set_outcome\(\)](#) (*pygambit.gambit.Game method*), [48](#)
[set_player\(\)](#) (*pygambit.gambit.Game method*), [46](#)
[simpdiv_solve\(\)](#) (*in module pygambit.nash*), [83](#)
[strategies](#) (*pygambit.gambit.Game attribute*), [51](#)
[strategies](#) (*pygambit.gambit.Player attribute*), [53](#)
[Strategy](#) (*class in pygambit.gambit*), [39](#)
[strategy_regret\(\)](#) (*pygam-
bit.gambit.MixedStrategyProfile method*),
[63](#)
[strategy_value\(\)](#) (*pygam-
bit.gambit.MixedStrategyProfile method*),
[62](#)
[strategy_value_deriv\(\)](#) (*pygam-
bit.gambit.MixedStrategyProfile method*),
[64](#)
[support_profile\(\)](#) (*pygambit.gambit.Game method*),
[59](#)

T

[title](#) (*pygambit.gambit.Game attribute*), [50](#)

U

[undominated_strategies_solve\(\)](#) (*in module
pygambit.supports*), [78](#)
[use_strategic](#) (*pygam-
bit.nash.NashComputationResult attribute*),
[79](#)

W

[write\(\)](#) (*pygambit.gambit.Game method*), [42](#)