Gambit Documentation

Release 15.1.1

The Gambit Project

May 12, 2017

Contents

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.

Using and citing Gambit

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:

> McKelvey, Richard D., McLennan, Andrew M., and Turocy, Theodore L. (2014). Gambit: Software Tools for Game Theory, Version 15.1.1. http://www.gambit-project.org.

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

Table of Contents

An overview of Gambit

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.

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.

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 Un*x (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.

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.

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.

Downloading Gambit

Gambit operates on an annual release cycle roughly mirroring the (northern hemisphere) academic year. A new version is promoted to stable/teaching each August; the major version number is equal to the last two digits of the year in which the version becomes stable.

This document covers Gambit 15, which is the current stable version as of October 2015. The most recent release is 15.1.1. You can download it from Sourceforge. Full source code is available, as are precompiled binaries for Microsoft Windows and Mac OS X 10.8.

Older versions of Gambit can be downloaded from http://sourceforge.net/projects/gambit/files. Support for older versions is limited.

Community

The following mailing lists are available for those interested in the use and further development of Gambit:

gambit-announce@lists.sourceforge.net Announcement-only mailing list for notifications of new releases of Gambit.

gambit-users@lists.sourceforge.net General discussion forum for teaching and research users of Gambit.

gambit-devel@lists.sourceforge.net Discussion for those interested in devleoping or extending Gambit, or using Gambit source code in other applications.

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.

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 API*.

General concepts

General layout of the main window
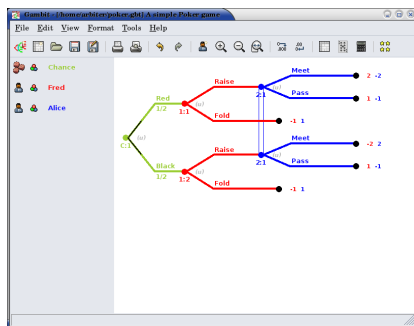
 The frame presenting a game consists of two principal panels. The main panel, to the right, displays the game graphically; in this case, showing the game tree of a simple one-card poker game. To the left is the player panel, which lists the players in the game; here, Fred and Alice are the players. Note 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*.

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.

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.

Extensive games

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

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.

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.

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.

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.

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.

This is an operation that is easier to see than the explain. See the poker tutorial (flash version; PDF version) for an application of the revelation operation in conjunction with the tree-copy operation.

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.

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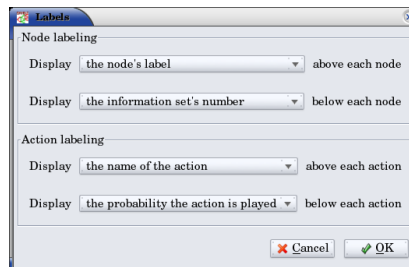
To remove an outcome from a node, click on the node, and select *Edit → Remove outcome*.

Formatting and labeling the tree

Gambit offers some options for customizing the display of game trees.

Labels on nodes and branches

The information displayed at the nodes and on the branches of the tree can be configured by selecting *Format →*

Labels, which displays the *tree labels* dialog. Above and below each node, the following information can be displayed:

No label The space is left blank.

The node's label The text label assigned to the node. (This is the default labeling above each node.)

The player's name The name of the player making the move at the node.

The information set's label The name of the information set to which the node belongs.

The information set's number A unique identifier of the information set, in the form player number:information set number. (This is the default labeling below each node.)

The realization probability The probability the node is reached. (Only displayed when a behavior strategy is selected to be displayed on the tree.)

The belief probability The probability a player assigns to being at the node, conditional on reaching the information set. (Only displayed when a behavior strategy is selected to be displayed on the tree.)

The payoff of reaching the node The expected payoff to the player making the choice at the node, conditional on reaching the node. (Only displayed when a behavior strategy is selected to be displayed on the tree.)

Above and below each branch, the following information can be displayed:

No label The space is left blank.

The name of the action The name of the action taken on the branch. (This it the default labeling above the branch.)

The probability the action is played For chance actions, the probability the branch is taken; this is always displayed. For player actions, the probability the action is taken in the selected profile (only displayed when a behavior strategy is selected to be displayed on the tree). In some cases, behavior strategies do not fully specify behavior sufficiently far off the equilibrium path; in such cases, an asterisk is shown for such action probabilities. (This is the default labeling below each branch.)

The value of the action The expected payoff to the player of taking the action, conditional on reaching the information set. (Only displayed when a behavior strategy is selected to be displayed on the tree.)

Controlling the layout of the tree

Gambit implements an automatic system for layout out game trees, which provides generally good results for most games. These can be adjusted by selecting *Format → Layout*. The layout parameters are organized on three tabs.
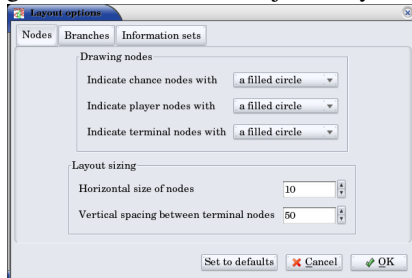
 The first tab, labeled *Nodes*, controls the size, location, and rendering of nodes in the tree. Nodes can be indicated using one of five tokens: a horizontal line (the "traditional" Gambit style from previous versions), a box, a diamond, an unfilled circle, and a filled circle). These can be set independently to distinguish chance and terminal nodes from player nodes.
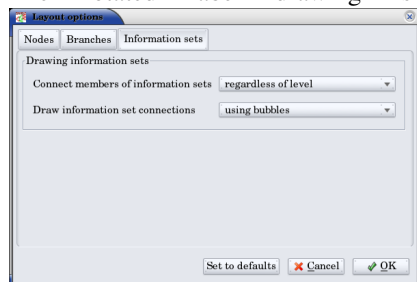
The sizing of nodes can be configured for best results. Gambit styling from previous versions used the horizontal line tokens with relatively long lines; when using the other tokens, smaller node sizes often look better.
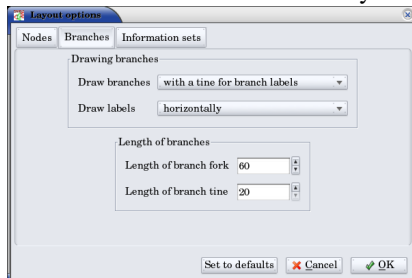
 The layout algorithm is based upon identifying the location of terminal nodes. The vertical spacing between these nodes can be set; making this value larger will tend to give the tree a larger vertical extent.

The second tab, labeled *Branches*, controls the display of the branches of the tree. The traditional Gambit way of drawing branches is a "fork-tine" approach, in which there is a flat part at the end of each branch at which labels are displayed. Alternatively, branches can be drawn directly between nodes by setting *Draw branches* to using straight lines between nodes. With this setting, labels are now displayed at points along the (usually) diagonal branches. Labels are usually shown horizontally; however, they can be drawn rotated parallel to the branches by setting *Draw labels* to rotated.

The rotated label drawing is experimental, and does not always look good on screen. The length used for branches and their tines, if drawn, can be configured. Longer branch and tine lengths give more space for longer labels to be drawn, at the cost of giving the tree a larger horizontal extent.

Finally, display of the information sets in the game is configured under the tab labeled *Information sets*. Members of information sets are by default connected using a "bubble" similar to that drawn in textbook diagrams of games. The can be modified to use a single line to connect nodes in the same information set. In conjunction with using lines for nodes, this can sometimes lead to a more compact representation of a tree where there are many information sets at the same horizontal location.

The layout of the tree may be such that members of the same information set appear at different horizontal locations in the tree. In such a case, by default, Gambit draws a horizontal arrow pointing rightward or leftward to indicate the continuation of the information set, as illustrated in the diagram nearby. These connections can be disabled by setting *Connect members of information sets* to *only when on the same level*. In addition, drawing information set indicators can be disabled entirely by setting this to invisibly (don't draw indicators).
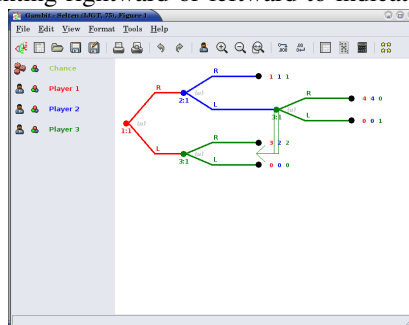
Selecting fonts and colors

To select the font used to draw the labels in the tree, select *Format → Font*. The standard font selection dialog for the operating system is displayed, showing the fonts available on the system. Since available fonts vary across systems, when opening a workbook on a system different from the system on which it was saved, Gambit tries to match the font style as closely as possible when the original font is not available.

The color-coding for each player can be changed by clicking on the color icon to the left of the corresponding player.

Strategic games

Gambit has full support for constructing and manipulating arbitrary N-player strategic (also known as normal form) games.
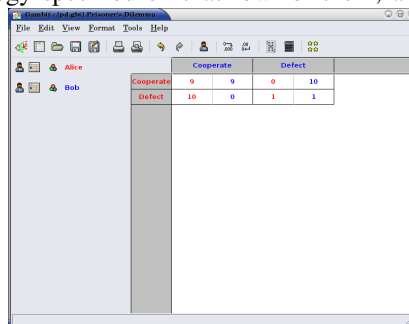
For extensive games, Gambit automatically computes the corresponding reduced strategic game. To view the reduced strategic game corresponding to an extensive game, select *View → Strategic game* or click the strategic game table icon on the toolbar.

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.

Navigating a strategic game

Gambit displays a strategic game in table form. All players are assigned to be either row players or column players, and the payoffs for each entry in the strategic game table correspond to the payoffs corresponding to the situation in which all the row players play the strategy specified on that row for them, and all the column players play the
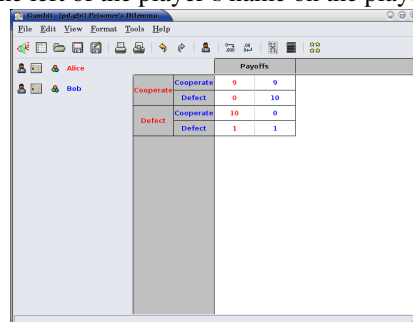

strategy specified on that column for them. For games with two players, this presentation is by default configured to be similar to the standard presenation of strategic games as tables, in which one player is assigned to be the "row" player and the other the "column" player. However, Gambit permits a more flexible assignment, in which multiple players can be assigned to the rows and multiple players to the columns. This is of particular use for games with more than two players. In print, a three-player strategic game is usually presented as a collection of tables, with one player choosing the row, the second the column, and the third the table. Gambit presents such games by hierarchially listing the strategies of one or more players on both rows and columns.

The hierarchical presentation of the table is similar to that of a contingency table in a spreadsheet application. Here, Alice, shown in red, has her strategies listed on the rows of the table, and Bob, shown in blue, has his strategies listed on the columns of the table.

The assignment of players to row and column roles is fully customizable. To change the assignment of a player, drag the person icon appearing to the left of the player's name on the player toolbar to either of the areas in the payoff table


displaying the strategy labels. For example, dragging the player icon from the left of Bob's name in the list of players and dropping it on the right side of Alice's strategy label column changes the display of the game as in Here, the strategies are shown in a hierarchical format, enumerating the outcomes of the game first by Alice's (red) strategy choice, then by Bob's (blue) strategy choice.

Alternatively, the game can be displayed by listing the outcomes with Bob's strategy choice first, then Alice's. Drag Bob's player icon and drop it on the left side of Alice's strategy choices, and the game display changes to organize the

outcomes first by Bob's action, then by Alice's.

The same dragging operation can be used to assign players to the columns. Assigning multiple players to the columns gives the same hierarchical presentation of those players' strategies. Dropping a player above another player's strategy labels assigns him to a higher level of the column player hierarchy; dropping a player below another player's strategy
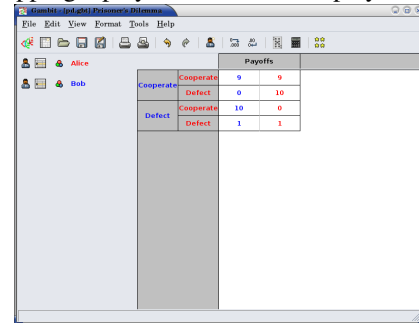
labels assigns him to a lower level of the column player hierarchy. As the assignment of players in the row and column hierarchies changes, the ordering of the payoffs in each cell of the table also changes. In all cases, the color-coding of the entries identifies the player to whom each payoff corresponds. The ordering convention is chosen so that for a two player game in which one player is a row player and the other a column player, the row player's payoff is shown first, followed by the column player, which is the most common convention in print.

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.

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.

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.

Dominated actions in extensive game

In extensive games, the dominance toolbar controls the elimination of actions which are conditionally dominated.
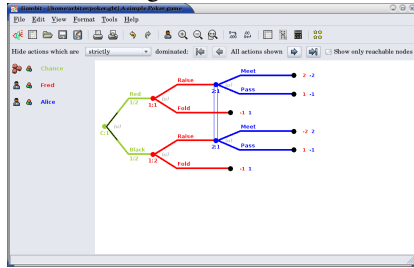
Actions may be eliminated based on two criteria:

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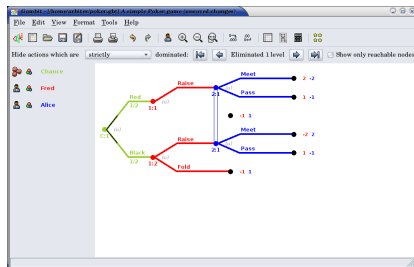
For example, in the poker game, it is strictly dominated for Fred to choose Fold after Red. Clicking the next level icon removes the dominated action from the game display.

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
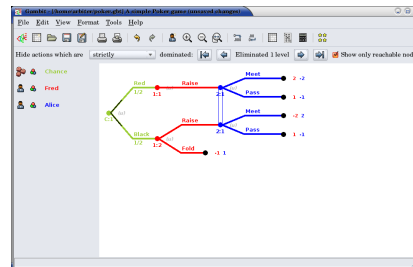

previous level icon moves up one level of elimination. The elimination of multiple levels can be automated using the fast forward icon , which iteratively eliminates dominated actions until no further actions can be eliminated. The rewind icon restores the display to the full game.

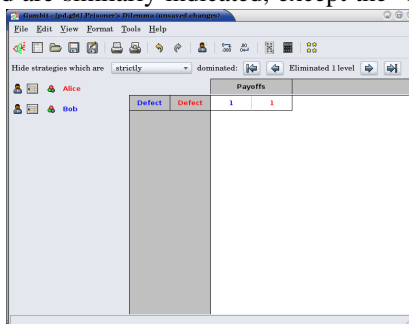## Dominated strategies in strategic games

The dominance toolbar operates in strategic games in the same way as the in the extensive game. Strategies can be
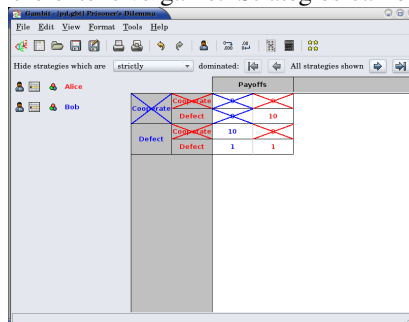
eliminated iteratively based on whether they are strictly or weakly dominated.
When the dominance toolbar is shown, the strategic game table contains indicators of strategies that are dominated. In the prisoner's dilemma, the Cooperate strategy is strictly dominated for both players. This strict dominance is indicated by the solid "X" drawn across the corresponding strategy labels for both players. In addition, the payoffs corresponding to the dominated strategies are also drawn with a solid "X" across them. Thus, any contingency in the table containing at least one "X" is a contingency that can only be reached by at least one player playing a strategy that is dominated.

Strategies that are weakly dominated are similarly indicated, except the "X" shape is drawn using a thinner, dashed

line instead of the thick, solid line. Clicking the next level icon removes the strictly dominated strategies from the display.
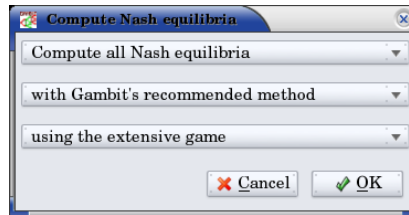
Computing Nash equilibria

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.

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.
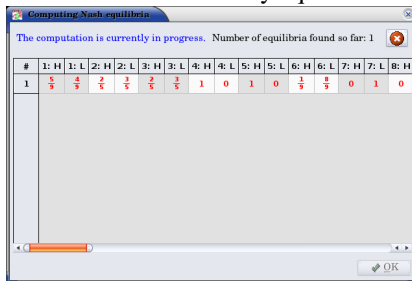
Gambit provides guidance on the options for computing Nash equilibria in a dialog. The methods applicable to a particular game depend on three criteria: the number of equilibria to compute, whether the computation is to be done on the extensive or strategic games, and on details of the game, such as 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 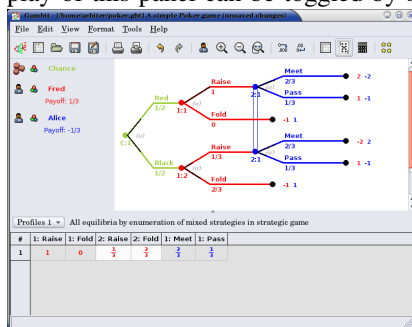details on these methods are contained in *Command-line tools*. Finally, for extensive games, there is an option of whether to use the extensive or strategic game for computation. In general, computation using the extensive game is preferred, since it is often a significantly more compact representation of the strategic chareceteristics 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.

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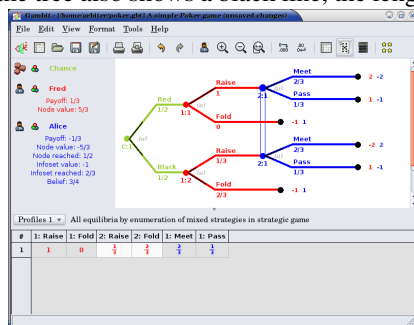
with which the action is played. Clicking on any node in the tree displays additional information about the profile at that node. The player panel displays information relevant to the selected node, including the payoff to all players conditional on reaching the node, as well as information about Alice's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

Computing quantal response equilibria

Gambit provides methods for computing the logit quantal response equilibrium correspondence for extensive games *[McKPal98]* and strategic games *[McKPal95]*, using the tracing method of *[Tur05]*.

To compute the correspondence, select *Tools → Qre*. If viewing an extensive game, the agent quantal response equilibrium correspondence is computed; if viewing a strategic game (including the reduced strategic game derived from an extensive game), the correspondence is computed in mixed strategies.

The computed correspondence values can be saved to a CSV (comma- separated values) file by clicking the button labeled *Save correspondence to .csv file*. This format is suitable for reading by a spreadsheet or graphing application.

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
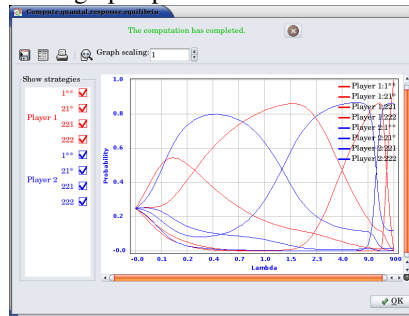
the scaling give more space to higher values of lambda. The strategies graphed are indicated in the panel at the left of the window. Clicking on the checkbox next to a strategy toggles whether it is displayed in the graph.

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.

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.

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.

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.

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 probabilites 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.

`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 equilbrium 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 15.1.1, Copyright (C) 1994-2014, 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 15.1.1, Copyright (C) 1994-2014, 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 15.1.1, Copyright (C) 1994-2014, 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
```

gambit-enumpoly: Compute equilibria of a game using polynomial systems of equations

gambit-enumpoly reads a game on standard input and computes Nash equilibria by solving systems of polynomial equations and inequalities.

This program searches for all Nash equilibria in a strategic game using a support enumeration approach. This approach computes all the supports which could, in principle, be the support of a Nash equilibrium, and then searches for a totally mixed equilibrium on that support by solving a system of polynomial equalities and inequalities formed by the Nash equilibrium conditions. The ordering of the supports is done in such a way as to maximize use of previously computed information, making it suited to the computation of all Nash equilibria.

When the verbose switch *-v* is used, the program outputs each support as it is considered. The supports are presented as a comma-separated list of binary strings, where each entry represents one player. The digit 1 represents a strategy which is present in the support, and the digit 0 represents a strategy which is not present. Each candidate support is printed with the label "candidate,".

Note that the subroutine to compute a solution to the system of polynomial equations and inequalities will fail in degenerate cases. When the verbose switch *-v* is used, these supports are identified on standard output with the label "singular,". It is possible that there exist equilibria, often a connected component of equilibria, on these singular supports.

-d

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

-h

> Prints a help message listing the available options.

-H

> By default, the program uses an enumeration method designed to visit as few supports as possible in searching for all equilibria. With this switch, the program uses a heuristic search method based on Porter, Nudelman, and Shoham *[PNS04]*, which is designed to minimize the time until the first equilibrium is found. This switch only has an effect when solving strategic games.

-S

> By default, the program uses behavior strategies for extensive games; this switch instructs the program to use reduced strategic game strategies for extensive games. (This has no effect for strategic games, since a strategic game is its own reduced strategic game.)

-q

> Suppresses printing of the banner at program launch.

-v

> Sets verbose mode. In verbose mode, supports are printed on standard output with the label "candidate" as they are considered, and singular supports are identified with the label "singular." By default, no information about supports is printed.

Computing equilbria of the extensive game `e01.efg`, the example in Figure 1 of Selten (International Journal of Game Theory, 1975) sometimes called "Selten's horse":

```
$ gambit-enumpoly e01.efg
Compute Nash equilibria by solving polynomial systems
Gambit version 15.1.1, Copyright (C) 1994-2014, The Gambit Project
Heuristic search implementation Copyright (C) 2006, Litao Wei
This is free software, distributed under the GNU GPL

NE,0.000000,1.000000,0.333333,0.666667,1.000000,0.000000
NE,1.000000,0.000000,1.000000,0.000000,0.250000,0.750000
NE,1.000000,0.000000,1.000000,0.000000,0.000000,0.000000
NE,0.000000,1.000000,0.000000,0.000000,1.000000,0.000000
```

`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 equilbria are identified, the program computes the convex sets which make up the set of equilibria. The program then additionally outputs each convex set, prefixed by convex-N , where N indexes the set. The set of all equilibria, then, is the union of these convex sets.

-h

> Prints a help message listing the available options.

-q

> Suppresses printing of the banner at program launch.

-L

> Use lrslib by David Avis to carry out the enumeration process. This is an experimental feature that has not been widely tested.

Computing the equilibria, in mixed strategies, of `e02.nfg`, the reduced strategic form of the example in Figure 2 of Selten (International Journal of Game Theory, 1975):

```
$ gambit-enummixed e02.nfg
Compute Nash equilibria by enumerating extreme points
Gambit version 15.1.1, Copyright (C) 1994-2014, 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-enummixed -c e02.nfg
Compute Nash equilibria by enumerating extreme points
Gambit version 15.1.1, Copyright (C) 1994-2014, 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
convex-1,1,0,0,1/2,1/2
convex-1,1,0,0,1,0
```

`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 a wrapper around the Gametracer 0.2 implementation by Ben Blum and Christian Shelton.

-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).

-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 15.1.1, Copyright (C) 1994-2014, The Gambit Project
This is free software, distributed under the GNU GPL

NE,1,0,2.99905e-12,0.5,0.5
```

Note: This is an experimental program and has not been extensively tested.

See also:

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

`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 a wrapper around the Gametracer 0.2 implementation by Ben Blum and Christian Shelton.

-d

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

-h

Prints a help message listing the available options.

-q

Suppresses printing of the banner at program launch.

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 15.1.1, Copyright (C) 1994-2014, The Gambit Project
This is free software, distributed under the GNU GPL

NE,1.000000,0.000000,0.000000,1.000000,0.000000
```

Note: This is an experimental program and has not been extensively tested.

See also:

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

`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 equilbrium 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 15.1.1, Copyright (C) 1994-2014, The Gambit Project
This is free software, distributed under the GNU GPL

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

`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 equilbrium 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 15.1.1, Copyright (C) 1994-2014, The Gambit Project
This is free software, distributed under the GNU GPL

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

`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 function 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.

-d

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

-n

> Specify the number of starting points to randomly generate.

-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 15.1.1, Copyright (C) 1994-2014, The Gambit Project
This is free software, distributed under the GNU GPL

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

`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).

-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 15.1.1, Copyright (C) 1994-2014, The Gambit Project
This is free software, distributed under the GNU GPL

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

`gambit-logit`: Compute quantal response equilbria

`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.

-d

>   Express all output using decimal representations with the specified number of digits. The default is *-d 6*.

-s

>   Sets the initial step size for the predictor phase of the tracing procedure. The default value is .03. The step size is specified in terms of the arclength along the branch of the correspondence, and not the size of the step measured in terms of lambda. So, for example, if the step size is currently .03, but the position of the strategy profile on the branch is changing rapidly with lambda, then lambda will change by much less then .03 between points reported by the program.

-a

>   Sets the maximum acceleration of the step size during the tracing procedure. This is interpreted as a multiplier. The default is 1.1, which means the step size is increased or decreased by no more than ten percent of its current value at every step. A value close to one would keep the step size (almost) constant at every step.

-m

>   Stop when reaching the specified value of the parameter lambda. By default, the tracing stops when lambda reaches 1,000,000, which is usually suitable for computing a good approximation to a Nash equilibrium. For applications, such as to laboratory experiments, where the behavior of the correspondence for small values of lambda is of interest and the asymptotic behavior is not relevant, setting MAXLAMBDA to a much smaller value may be indicated.

-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 15.1.1, Copyright (C) 1994-2014, The Gambit Project
This is free software, distributed under the GNU GPL

0.000000,0.333333,0.333333,0.333333,0.5,0.5
0.022853,0.335873,0.328284,0.335843,0.501962,0.498038
0.047978,0.338668,0.322803,0.33853,0.504249,0.495751
0.075600,0.341747,0.316863,0.34139,0.506915,0.493085
0.105965,0.345145,0.310443,0.344413,0.510023,0.489977
0.139346,0.348902,0.303519,0.347578,0.51364,0.48636

...

735614.794714,1,0,4.40659e-11,0.500016,0.499984
809176.283787,1,0,3.66976e-11,0.500015,0.499985
890093.921767,1,0,3.05596e-11,0.500014,0.499986
979103.323545,1,0,2.54469e-11,0.500012,0.499988
1077013.665501,1,0,2.11883e-11,0.500011,0.499989
```

`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 15.1.1, Copyright (C) 1994-2014, 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 15.1.1, Copyright (C) 1994-2014, 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}
```

Python interface to Gambit library

Gambit provides a Python interface for programmatic manipulation of games. This section documents this interface, which is under active development. Refer to the *instructions for building the Python interface* to compile and install the Python extension.

A tutorial introduction

Building an extensive game

The function *Game.new_tree()* creates a new, trivial extensive game, with no players, and only a root node:

```
In [1]: import gambit

In [2]: g = gambit.Game.new_tree()

In [3]: len(g.players)
Out[3]: 0
```

The game also has no title. The *title* attribute provides access to a game's title:

```
In [4]: str(g)
Out[4]: "<Game ''>"
```

```
In [5]: g.title = "A simple poker example"

In [6]: g.title
Out[6]: 'A simple poker example'

In [7]: str(g)
Out[7]: "<Game 'A simple poker example'>"
```

The *players* attribute of a game is a collection of the players. As seen above, calling `len()` on the set of players gives the number of players in the game. Adding a *Player* to the game is done with the `add()` member of *players*:

```
In [8]: p = g.players.add("Alice")

In [9]: p
Out[9]: <Player [0] 'Alice' in game 'A simple poker example'>
```

Each *Player* has a text string stored in the *label* attribute, which is useful for human identification of players:

```
In [10]: p.label
Out[10]: 'Alice'
```

Game.players can be accessed like a Python list:

```
In [11]: len(g.players)
Out[11]: 1

In [12]: g.players[0]
Out[12]: <Player [0] 'Alice' in game 'A simple poker example'>

In [13]: g.players
Out[13]: [<Player [0] 'Alice' in game 'A simple poker example'>]
```

Building a strategic game

Games in strategic form are created using *Game.new_table()*, which takes a list of integers specifying the number of strategies for each player:

```
In [1]: g = gambit.Game.new_table([2,2])

In [2]: g.title = "A prisoner's dilemma game"

In [3]: g.players[0].label = "Alphonse"

In [4]: g.players[1].label = "Gaston"

In [5]: g
Out[5]:
NFG 1 R "A prisoner's dilemma game" { "Alphonse" "Gaston" }

{ { "1" "2" }
{ "1" "2" }
}
""

{
```

```
}
0 0 0 0
```

The *strategies* collection for a *Player* lists all the strategies available for that player:

```
In [6]: g.players[0].strategies
Out[6]: [<Strategy [0] '1' for player 'Alphonse' in game 'A
prisoner's dilemma game'>,
        <Strategy [1] '2' for player 'Alphonse' in game 'A prisoner's dilemma game'>]

In [7]: len(g.players[0].strategies)
Out[7]: 2

In [8]: g.players[0].strategies[0].label = "Cooperate"

In [9]: g.players[0].strategies[1].label = "Defect"

In [10]: g.players[0].strategies
Out[10]: [<Strategy [0] 'Cooperate' for player 'Alphonse' in game 'A
prisoner's dilemma game'>,
        <Strategy [1] 'Defect' for player 'Alphonse' in game 'A prisoner's dilemma␣
→game'>]
```

The outcome associated with a particular combination of strategies is accessed by treating the game like an array. For a game g, g[i, j] is the outcome where the first player plays his i th strategy, and the second player plays his j th strategy. Payoffs associated with an outcome are set or obtained by indexing the outcome by the player number. For a prisoner's dilemma game where the cooperative payoff is 8, the betrayal payoff is 10, the sucker payoff is 2, and the noncooperative (equilibrium) payoff is 5:

```
In [11]: g[0,0][0] = 8

In [12]: g[0,0][1] = 8

In [13]: g[0,1][0] = 2

In [14]: g[0,1][1] = 10

In [15]: g[1,0][0] = 10

In [16]: g[1,1][1] = 2

In [17]: g[1,0][1] = 2

In [18]: g[1,1][0] = 5

In [19]: g[1,1][1] = 5
```

Alternatively, one can use *Game.from_arrays()* in conjunction with numpy arrays to construct a game with desired payoff matrices more directly, as in:

```
In [20]: m = numpy.array([ [ 8, 2 ], [ 10, 5 ] ], dtype=gambit.Rational)

In [21]: g = gambit.Game.from_arrays(m, numpy.transpose(m))
```

Reading a game from a file

Games stored in existing Gambit savefiles in either the .efg or .nfg formats can be loaded using *Game.read_game()*:

```
In [1]: g = gambit.Game.read_game("e02.nfg")

In [2]: g
Out[2]:
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
```

Iterating the pure strategy profiles in a game

Each entry in a strategic game corresponds to the outcome arising from a particular combination fo pure strategies played by the players. The property *Game.contingencies* is the collection of all such combinations. Iterating over the contingencies collection visits each pure strategy profile possible in the game:

```
In [1]: g = gambit.Game.read_game("e02.nfg")

In [2]: list(g.contingencies)
Out[2]: [[0, 0], [0, 1], [1, 0], [1, 1], [2, 0], [2, 1]]
```

Each pure strategy profile can then be used to access individual outcomes and payoffs in the game:

```
In [3]: for profile in g.contingencies:
   ...:     print profile, g[profile][0], g[profile][1]
   ...:
[0, 0] 1 1
[0, 1] 1 1
[1, 0] 0 2
[1, 1] 0 3
[2, 0] 0 2
[2, 1] 2 0
```

Mixed strategy and behavior profiles

A *MixedStrategyProfile* object, which represents a probability distribution over the pure strategies of each player, is constructed using *Game.mixed_strategy_profile()*. Mixed strategy profiles are initialized to uniform randomization over all strategies for all players.

Mixed strategy profiles can be indexed in three ways.

1. Specifying a strategy returns the probability of that strategy being played in the profile.

2. Specifying a player returns a list of probabilities, one for each strategy available to the player.

3. Profiles can be treated as a list indexed from 0 up to the number of total strategies in the game minus one.

This sample illustrates the three methods:

```
In [1]: g = gambit.Game.read_game("e02.nfg")

In [2]: p = g.mixed_strategy_profile()

In [3]: list(p)
Out[3]: [0.33333333333333331, 0.33333333333333331, 0.33333333333333331, 0.5, 0.5]

In [4]: p[g.players[0]]
Out[4]: [0.33333333333333331, 0.33333333333333331, 0.33333333333333331]

In [5]: p[g.players[1].strategies[0]]
Out[5]: 0.5
```

The expected payoff to a player is obtained using *MixedStrategyProfile.payoff()*:

```
In [6]: p.payoff(g.players[0])
Out[6]: 0.66666666666666663
```

The standalone expected payoff to playing a given strategy, assuming all other players play according to the profile, is obtained using *MixedStrategyProfile.strategy_value()*:

```
In [7]: p.strategy_value(g.players[0].strategies[2])
Out[7]: 1.0
```

A *MixedBehaviorProfile* object, which represents a probability distribution over the actions at each information set, is constructed using *Game.mixed_behavior_profile()*. Behavior profiles are initialized to uniform randomization over all actions at each information set.

Mixed behavior profiles are indexed similarly to mixed strategy profiles, except that indexing by a player returns a list of lists of probabilities, containing one list for each information set controlled by that player:

```
In [1]: g = gambit.Game.read_game("e02.efg")

In [2]: p = g.mixed_behavior_profile()

In [3]: list(p)
Out[3]: [0.5, 0.5, 0.5, 0.5, 0.5, 0.5]

In [5]: p[g.players[0]]
Out[5]: [[0.5, 0.5], [0.5, 0.5]]

In [6]: p[g.players[0].infosets[0]]
Out[6]: [0.5, 0.5]

In [7]: p[g.players[0].infosets[0].actions[0]]
Out[7]: 0.5
```

For games with a tree representation, a *MixedStrategyProfile* can be converted to its equivalent *MixedBehaviorProfile* by calling *MixedStrategyProfile.as_behavior()*. Equally, a *MixedBehaviorProfile* can be converted to an equivalent *MixedStrategyProfile* using *MixedBehaviorProfile.as_strategy()*.

Computing Nash equilibria

Interfaces to algorithms for computing Nash equilibria are collected in the module `gambit.nash`. There are two choices for calling these algorithms: directly within Python, or via the corresponding Gambit *command-line tool*.

Calling an algorithm directly within Python has less overhead, which makes this approach well-suited to the analysis of smaller games, where the expected running time is small. In addition, these interfaces may offer more fine-grained control of the behavior of some algorithms.

Calling the Gambit command-line tool launches the algorithm as a separate process. This makes it easier to abort during the run of the algorithm (preserving where possible the equilibria which have already been found), and also makes the program more robust to any internal errors which may arise in the calculation.

Calling command-line tools

The interface to each command-line tool is encapsulated in a class with the word "External" in the name. These operate by creating a subprocess, which calls the corresponding Gambit *command-line tool*. Therefore, a working Gambit installation needs to be in place, with the command-line tools located in the executable search path.

Method	Python class
gambit-enumpure	ExternalEnumPureSolver
gambit-enummixed	ExternalEnumMixedSolver
gambit-lp	ExternalLPSolver
gambit-lcp	ExternalLCPSolver
gambit-simpdiv	ExternalSimpdivSolver
gambit-gnm	ExternalGlobalNewtonSolver
gambit-enumpoly	ExternalEnumPolySolver
gambit-liap	ExternalLyapunovSolver
gambit-ipa	ExternalIteratedPolymatrixSolver
gambit-logit	ExternalLogitSolver

For example, consider the game `e02.nfg` from the set of standard Gambit examples. This game has a continuum of equilibria, in which the first player plays his first strategty with probability one, and the second player plays a mixed strategy, placing at least probability one-half on her first strategy:

```
In [1]: g = gambit.Game.read_game("e02.nfg")

In [2]: solver = gambit.nash.ExternalEnumPureSolver()

In [3]: solver.solve(g)
Out[3]: [[1.0, 0.0, 0.0, 1.0, 0.0]]

In [4]: solver = gambit.nash.ExternalEnumMixedSolver()

In [5]: solver.solve(g)
Out[5]: [[1.0, 0.0, 0.0, 1.0, 0.0], [1.0, 0.0, 0.0, 0.5, 0.5]]

In [6]: solver = gambit.nash.ExternalLogitSolver()

In [7]: solver.solve(g)
Out[7]: [[0.99999999997881173, 0.0, 2.1188267679986399e-11, 0.50001141005647654, 0.
→49998858994352352]]
```

In this example, the pure strategy solver returns the unique equilibrium in pure strategies. Solving using **gambit-enummixed** gives two equilibria, which are the extreme points of the set of equilibria. Solving by tracing the quantal response equilibrium correspondence produces a close numerical approximation to one equilibrium; in

fact, the equilibrium which is the limit of the principal branch is the one in which the second player randomizes with equal probability on both strategies.

When a game's representation is in extensive form, these solvers default to using the version of the algorithm which operates on the extensive game, where available, and returns a list of *gambit.MixedBehaviorProfile* objects. This can be overridden when calling solve() via the use_strategic parameter:

```
In [1]: g = gambit.Game.read_game("e02.efg")

In [2]: solver = gambit.nash.ExternalLCPSolver()

In [3]: solver.solve(g)
Out[3]: [<NashProfile for 'Selten (IJGT, 75), Figure 2': [1.0, 0.0, 0.5, 0.5, 0.5, 0.
↪5]>]

In [4]: solver.solve(g, use_strategic=True)
Out[4]: [<NashProfile for 'Selten (IJGT, 75), Figure 2': [1.0, 0.0, 0.0, 1.0, 0.0]>]
```

As this game is in extensive form, in the first call, the returned profile is a *MixedBehaviorProfile*, while in the second, it is a *MixedStrategyProfile*. While the set of equilibria is not affected by whether behavior or mixed strategies are used, the equilibria returned by specific solution methods may differ, when using a call which does not necessarily return all equilibria.

Calling internally-linked libraries

Where available, versions of algorithms which have been linked internally into the Python library are generally called via convenience functions. The following table lists the algorithms available via this approach.

Method	Python function
gambit-enumpure	*gambit.nash.enumpure_solve()*
gambit-lp	*gambit.nash.lp_solve()*
gambit-lcp	*gambit.nash.lcp_solve()*

Parameters are available to modify the operation of the algorithm. The most common ones are use_strategic, to indicate the use of a strategic form version of an algorithm where both extensive and strategic versions are available, and rational, to indicate computation using rational arithmetic, where this is an option to the algorithm.

For example, taking again the game e02.efg as an example:

```
In [1]: g = gambit.Game.read_game("e02.efg")

In [2]: gambit.nash.lcp_solve(g)
Out[2]: [[1.0, 0.0, 0.5, 0.5, 0.5, 0.5]]

In [3]: gambit.nash.lcp_solve(g, rational=True)
Out[3]: [[Fraction(1, 1), Fraction(0, 1), Fraction(1, 2), Fraction(1, 2), Fraction(1,
↪2), Fraction(1, 2)]]

In [4]: gambit.nash.lcp_solve(g, use_strategic=True)
Out[4]: [[1.0, 0.0, 0.0, 1.0, 0.0]]

In [5]: gambit.nash.lcp_solve(g, use_strategic=True, rational=True)
Out[5]: [[Fraction(1, 1), Fraction(0, 1), Fraction(0, 1), Fraction(1, 1), Fraction(0,
↪1)]]
```

API documentation

Game representations

class gambit.**Game**

An object representing a game, in extensive or strategic form.

classmethod new_tree()

Creates a new *Game* consisting of a trivial game tree, with one node, which is both root and terminal, and no players.

classmethod new_table(*dim*)

Creates a new *Game* with a strategic representation.

> **Parameters dim** – A list specifying the number of strategies for each player.

classmethod from_arrays(**arrays*)

Creates a new *Game* with a strategic representation. Each entry in arrays is a numpy array giving 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.

classmethod read_game(*fn*)

Constructs a game from its serialized representation in a file. See *Game representation formats* for details on recognized formats.

> **Parameters fn** (`file`) – The path to the file to open
>
> **Raises IOError** – if the file cannot be opened, or does not contain a valid game representation

classmethod parse_game(*s*)

Constructs a game from its seralized representation in a string. See *Game representation formats* for details on recognized formats.

> **Parameters s** (`str`) – The string containing the serialized representation
>
> **Raises IOError** – if the string does not contain a valid game representation

is_tree

Returns `True` if the game has a tree representation.

title

Accesses the text string of the game's title.

comment

Accesses the text string of the game's comment.

actions

Returns a list-like object representing the actions defined in the game.

> **Raises** *gambit.UndefinedOperationError* – if the game does not have a tree representation.

infosets

Returns a list-like object representing the information sets defined in the game.

> **Raises** *gambit.UndefinedOperationError* – if the game does not have a tree representation.

players

Returns a *Players* collection object representing the players defined in the game.

strategies

Returns a list-like object representing the strategies defined in the game.

contingencies
>   Returns a collection object representing the collection of all possible pure strategy profiles in the game.

root
>   Returns the *Node* representing the root node of the game.

>>   **Raises** *UndefinedOperationError* if the game does not have a tree representation.

is_const_sum
>   Returns `True` if the game is constant sum.

is_perfect_recall
>   Returns `True` if the game is of perfect recall.

min_payoff
>   Returns the smallest payoff in any outcome of the game.

max_payoff
>   Returns the largest payoff in any outcome of the game.

__getitem__(*profile*)
>   Returns the *Outcome* associated with a profile of pure strategies.

>>   **Parameters** **profile** – A list of integers specifying the strategy number each player plays in the profile.

mixed_strategy_profile(*rational=False*)
>   Returns a mixed strategy profile *MixedStrategyProfile* over the game, initialized to uniform randomization for each player over his strategies. If the game has a tree representation, the mixed strategy profile is defined over the reduced strategic form representation.

>>   **Parameters** **rational** – If `True`, probabilities are represented using rational numbers; otherwise double-precision floating point numbers are used.

mixed_behavior_profile(*rational=False*)
>   Returns a behavior strategy profile *MixedBehaviorProfile* over the game, initialized to uniform randomization for each player over his actions at each information set.

>>   **Parameters** **rational** – If `True`, probabilities are represented using rational numbers; otherwise double-precision floating point numbers are used.

>>   **Raises** *UndefinedOperationError* – if the game does not have a tree representation.

write(*format='native'*)
>   Returns 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*.

class gambit.**StrategicRestriction**
>   A read-only view on a *Game*, defined by a subset of the strategies on the original game.

>   In addition to the members described here, a StrategicRestriction implements the interface of a *Game*, although operations which change the content of the game will raise an exception.

unrestrict()
> Returns the *Game* object on which the restriction was based.

Representations of play of games

The main responsibility of these classes is to capture information about a plan of play of a game, by one or more players.

class gambit.**StrategySupportProfile**
> A set-like object representing a subset of the strategies in a game. It incorporates the restriction that each player must have at least one strategy.

> **game**
> > Returns the *Game* on which the support profile is defined.

> **issubset**(*other*)
> > Returns True if this profile is a subset of *other*.
> >
> > > **Parameters other** (StrategySupportProfile) – another support profile

> **issuperset**(*other*)
> > Returns True if this profile is a superset of *other*.
> >
> > > **Parameters other** (StrategySupportProfile) – another support profile

> **restrict**()
> > Creates a *StrategicRestriction* object, which defines a restriction of the game in which only the strategies in this profile are present.

> **remove**(*strategy*)
> > Modifies the support profile by removing the specified strategy.
> >
> > > **Parameters strategy** (Strategy) – the strategy to remove
> > >
> > > **Raises** *UndefinedOperationError* – if attempting to remove the last strategy for a player

> **difference**(*other*)
> > Returns a new support profile containing all the strategies which are present in this profile, but not in *other*.
> >
> > > **Parameters other** (StrategySupportProfile) – another support profile

> **intersection**(*other*)
> > Returns a new support profile containing all the strategies present in both this profile and in *other*.
> >
> > > **Parameters other** (StrategySupportProfile) – another support profile

> **union**(*other*)
> > Returns a new support profile containing all the strategies present in this profile, in *other*, or in both.
> >
> > > **Parameters other** (StrategySupportProfile) – another support profile

class gambit.**MixedStrategyProfile**
> Represents a mixed strategy profile over a *Game*.

> **__getitem__**(*index*)
> > Returns a slice of the profile based on the parameter index.
> >
> > > •If index is a *Strategy*, returns the probability with which that strategy is played in the profile.
> > >
> > > •If index is a *Player*, returns a list of probabilities, one for each strategy belonging to that player.
> > >
> > > •If index is an integer, returns the index th entry in the profile, treating the profile as a flat list of probabilities.

__setitem__(*strategy*, *prob*)
Sets the probability `strategy` is played in the profile to `prob`.

as_behavior()
Returns a behavior strategy profile `BehavProfile` associated to the profile.

> **Raises** *[gambit.UndefinedOperationError](#)* – if the game does not have a tree representation.

copy()
Creates a copy of the mixed strategy profile.

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

strategy_value(*strategy*)
Returns the expected payoff to choosing the strategy, if all other players play according to the profile.

strategy_values(*player*)
Returns the expected payoffs for a player's set of strategies if all other players play according to the profile.

liap_value()
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.

class gambit.**MixedBehaviorProfile**
Represents a behavior strategy profile over a *[Game](#)*.

__getitem__(*index*)
Returns a slice of the profile based on the parameter `index`.

- If `index` is a *[Action](#)*, returns the probability with which that action is played in the profile.

- If `index` is an *[Infoset](#)*, returns a list of probabilities, one for each action belonging to that information set.

- If `index` is a *[Player](#)*, returns a list of lists of probabilities, one list for each information set controlled by the player.

- If `index` is an integer, returns the `index` th entry in the profile, treating the profile as a flat list of probabilities.

__setitem__(*action*, *prob*)
Sets the probability `action` is played in the profile to `prob`.

as_strategy()
Returns a *[MixedStrategyProfile](#)* which is equivalent to the profile.

belief(*node*)
Returns the probability `node` is reached, given its information set was reached.

belief(*infoset*)
Returns a list of belief probabilities of each node in `infoset`.

copy()
Creates a copy of the behavior strategy profile.

payoff(*player*)
Returns the expected payoff to `player` if all players play according to the profile.

payoff(*action*)
Returns the expected payoff to choosing `action`, conditional on having reached the information set, if all other players play according to the profile.

payoff(*infoset*)

> Returns the expected payoff to the player who has the move at `infoset`, conditional on the information set being reached, if all players play according to the profile.

regret(*action*)

> Returns the regret associated to `action`.

realiz_prob(*infoset*)

> Returns the probability with which information set `infoset` is reached, if all players play according to the profile.

liap_value()

> 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.

Elements of games

These classes represent elements which exist inside of the definition of game.

class gambit.**Rational**

> New in version 15.0.0.
>
> Represents a rational number in specifying numerical data for a game, or in a computed strategy profile. This is implemented as a subclass of the Python standard library `fractions.Fraction`, with additional instrumentation for rendering in IPython notebooks.

class gambit.**Players**

> A collection object representing the players in a game.
>
> **len**()
>
> > Returns the number of players in the game.
>
> **__getitem__**(*i*)
>
> > Returns player number `i` in the game. Players are numbered starting with `0`.
>
> **chance**
>
> > Returns the player representing all chance moves in the game.
>
> **add**($\big[$*label=""*$\big]$)
>
> > Add a *Player* to the game. If label is specified, sets the text label for the player. In the case of extensive games this will create a new player with no moves. In the case of strategic form games it creates a player with one strategy. If the provided player label is shared by another player a warning will be returned.

class gambit.**Player**

> Represents a player in a *Game*.
>
> **game**
>
> > Returns the *Game* in which the player is.
>
> **label**
>
> > A text label useful for identification of the player.
>
> **number**
>
> > Returns the number of the player in the *Game*. Players are numbered starting with `0`.
>
> **is_chance**
>
> > Returns `True` if the player object represents the chance player.
>
> **infosets**
>
> > Returns a list-like object representing the information sets of the player.

> **strategies**
>> Returns a `gambit.Strategies` collection object representing the strategies of the player.
>
> **min_payoff**
>> Returns the smallest payoff for the player in any outcome of the game.
>
> **max_payoff**
>> Returns the largest payoff for the player in any outcome of the game.

class `gambit.`**`Infoset`**
> An information set for an extensive form game.
>
> **precedes**(*node*)
>> Returns `True` or `False` depending on whether the specified node precedes the information set in the extensive game.
>
> **reveal**(*player*)
>> Reveals the information set to a player.
>
> **actions**
>> Returns a `gambit.Actions` collection object representing the actions defined in this information set.
>
> **label**
>> A text label used to identify the information set.
>
> **is_chance**
>> Returns `True` or `False` depending on whether this information set is associated to the chance player.
>
> **members**
>> Returns the set of nodes associated with this information set.
>
> **player**
>> Returns the player object associated with this information set.

class `gambit.`**`Actions`**
> A collection object representing the actions available at an information set in a game.
>
> **len**()
>> Returns the number of actions for the player.
>
> **__getitem__**(*i*)
>> Returns action number `i`. Actions are numbered starting with `0`.
>
> **add**([*action=None*])
>> Add a `Action` to the list of actions of an information set.

class `gambit.`**`Action`**
> An action associated with an information set.
>
> **delete**()
>> Deletes this action from the game.
>>
>>> **Raises** `gambit.UndefinedOperationError` – when the action is the last one of its infoset.
>
> **precedes**(*node*)
>> Returns `True` if `node` precedes this action in the extensive game.
>
> **label**
>> A text label used to identify the action.
>
> **infoset**
>> Returns the information to which this action is associated.

---

prob
>   A settable property that represents the probability associated with the action. It can be a value stored as an int, *gambit.Rational*, or gambit.Decimal.

class gambit.**Strategies**
>   A collection object representing the strategies available to a player in a game.

len()
>   Returns the number of strategies for the player.

__getitem__(*i*)
>   Returns strategy number i. Strategies are numbered starting with 0.

add($\big[$*label=""* $\big]$)
>   Add a *Strategy* to the player's list of strategies.
>
>   >   **Raises TypeError** – if called on a game which has an extensive representation.

class gambit.**Strategy**
>   Represents a strategy available to a *Player*.

label
>   A text label useful for identification of the strategy.

class gambit.**Node**
>   Represents a node in a *Game*.

is_successor_of(*node*)
>   Returns True if the node is a successor of node.

is_subgame_root(*node*)
>   Returns True if the current node is a root of a proper subgame.

label
>   A text label useful for identification of the node.

is_terminal
>   Returns True if the node is a terminal node in the game tree.

children
>   Returns a collection of the node's children.

game
>   Returns the *Game* to which the node belongs.

infoset
>   Returns the *Infoset* associated with the node.

player
>   Returns the *Player* associated with the node.

parent
>   Returns the *Node* that is the parent of this node.

prior_action
>   Returns the action immediately prior to the node.

prior_sibling
>   Returns the *Node* that is prior to the node at the same level of the game tree.

next_sibling
>   Returns the *Node* that is the next node at the same level of the game tree.

outcome

Returns the *Outcome* that is associated with the node.

append_move (*infoset* [, *actions*])

Add a move to a terminal node, at the *gambit.Infoset* infoset. Alternatively, a *gambit.Player* can be passed as the information set, in which case the move is placed in a new information set for that player; in this instance, the number of actions at the new information set must be specified.

> **Raises**
>
> - **gambit.UndefinedOperationError** – when called on a non-terminal node.
>
> - **gambit.UndefinedOperationError** – when called with a *Player* object and no actions, or actions < 1.
>
> - **gambit.UndefinedOperationError** – when called with a *Infoset* object and with actions.
>
> - **gambit.MismatchError** – when called with objects from different games.

insert_move (*infoset* [, *actions*])

Insert a move at a node, at the *Infoset* infoset. Alternatively, a *Player* can be passed as the information set, in which case the move is placed in a new information set for that player; in this instance, the number of actions at the new information set must be specified. The newly-inserted node takes the place of the node in the game tree, and the existing node becomes the first child of the new node.

> **Raises**
>
> - **gambit.UndefinedOperationError** – when called with a *Player* object and no actions, or actions < 1.
>
> - **gambit.UndefinedOperationError** – when called with a *Infoset* object and with actions.
>
> - **gambit.MismatchError** – when called with objects from different games.

leave_infoset ()

Removes this node from its information set. If this node is the last of its information set, this method does nothing.

delete_parent ()

Deletes the parent node and its subtrees other than the one which contains this node and moves this node into its former parent's place.

delete_tree ()

Deletes the whole subtree which has this node as a root, except the actual node.

copy_tree (*node*)

Copies the subtree rooted at this node to node.

> **Raises** **gambit.MismatchError** – if both objects aren't in the same game.

move_tree (*node*)

Move the subtree rooted at this node to node.

> **Raises** **gambit.MismatchError** – if both objects aren't in the same game.

class gambit.**Outcomes**

A collection object representing the outcomes of a game.

len ()

Returns the number of outcomes in the game.

> **__getitem__**(*i*)
> > Returns outcome `i` in the game. Outcomes are numbered starting with `0`.
>
> **add**($\big[$*label=""*$\big]$)
> > Add a [*Outcome*](#) to the game. If label is specified, sets the text label for the outcome. If the provided outcome label is shared by another outcome a warning will be returned.

class gambit.**Outcome**
> Represents an outcome in a [*Game*](#).
>
> **delete**()
> > Deletes the outcome from the game.
>
> **label**
> > A text label useful for identification of the outcome.
>
> **__getitem__**(*player*)
> > Returns the payoff to `player` at the outcome. `player` may be a [*Player*](#), a string, or an integer. If a string, returns the payoff to the player with that string as its label. If an integer, returns the payoff to player number `player`.
>
> **__setitem__**(*player*, *payoff*)
> > Sets the payoff to the `pl` th player at the outcome to the specified `payoff`. Payoffs may be specified as integers or instances of gambit.Decimal or [*gambit.Rational*](#). Players may be specified as in [*__getitem__()*](#).

Representation of errors and exceptions

exception gambit.**MismatchError**
> A subclass of `ValueError` which is raised when attempting an operation among objects from different games.

exception gambit.**UndefinedOperationError**
> A subclass of `ValueError` which is raised when an operation which is not well-defined is attempted.

Computation of Nash equilibria

gambit.nash.**enumpure_solve**(*game*, *use_strategic=True*, *external=False*)
> Compute [*pure-strategy Nash equilibria*](#) of a game.
>
> > **Parameters**
> >
> > - **use_strategic** (*bool*) – Use the strategic form. If `False`, computes agent-form pure-strategy equilibria, which treat only unilateral deviations at an individual information set
> > - **external** (*bool*) – Call the external command-line solver instead of the internally-linked implementation

gambit.nash.**lcp_solve**(*game*, *rational=True*, *use_strategic=False*, *external=False*, *stop_after=None*, *max_depth=None*)
> Compute Nash equilibria of a two-player game using [*linear complementarity programming*](#).
>
> > **Parameters**
> >
> > - **rational** (*bool*) – Compute using rational precision (more precise, often much slower)
> > - **use_strategic** (*bool*) – Use the strategic form version even for extensive games
> > - **external** (*bool*) – Call the external command-line solver instead of the internally-linked implementation

- **stop_after** (*int*) – Number of equilibria to contribute (default is to compute until all reachable equilbria are found)

- **max_depth** (*int*) – Maximum recursion depth (default is no limit)

Raises RuntimeError – if game has more than two players.

gambit.nash.**lp_solve**(*game*, *rational=True*, *use_strategic=False*, *external=False*)
Compute Nash equilibria of a two-player constant-sum game using *linear programming*.

Parameters

- **rational** (*bool*) – Compute using rational precision (more precise, often much slower)

- **use_strategic** (*bool*) – Use the strategic form version even for extensive games

- **external** (*bool*) – Call the external command-line solver instead of the internally-linked implementation

Raises RuntimeError – if game has more than two players.

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 contributors: Ideas and suggestions for Gambit-related projects

Research on doing computation on finite games, and using numerical and algorithmic methods to analyze games, are areas of quite active research. There are a number of opportunities for programmers of all skill levels and backgrounds to contribute to improving and extending Gambit.

A number of such ideas are outlined in this section. Each project includes the required implementation environment, and a summary of the background prerequisites someone should have in order to take on the project successfully, in terms of mathematics, game theory, and software engineering.

Students who are interested in applying to participate in the Google Summer of Code program should also first read our introductory document and application template at http://www.gambit-project.org/application.txt.

For beginning contibutors - especially those who are interested in potentially applying to work on Gambit projects in future Google Summer of Code editions - there are a number of issues in the Gambit issue tracker tagged as "easy". These are excellent ways to get familiar with the Gambit codebase. Contributors who have completed one or more such easy tasks will have a significantly greater chance of being considered for possible GSoC work.

The Gambit source tree is managed using git. It is recommended to have some familiarity with how git works, or to be willing to learn. (It's not that hard, and once you do learn it, you'll wonder how you ever lived without it.)

This section lists project ideas pertaining to the Gambit library and desktop interface. There are additional project opportunities in the Game Theory Explorer web/cloud interface. These are listed separately here.

Refactor and update game representation library

The basic library (in `src/libgambit`) for representing games was written in the mid-1990s. As such, it predates many modern C++ features (including templates, STL, exceptions, Boost, and so on). There are a number of projects for taking existing functionality, refactoring it into separate components, and then enhancing what those components can do.

File formats for serializing games

Gambit supports a number of file formats for reading and writing games. There are traditional formats for extensive and strategic games. The graphical interface wraps these formats into an XML document which can store additional metadata about the game. The Game Theory Explorer also defines an XML format for storing games. And, from Gambit 14, there is a special file format for representing games in action graph format.

Separately, Gambit has a command-line tool for outputting games in HTML and LaTeX formats.

This project would refactor these features into a unified framework for reading and writing games in various formats. It would include:

- Migrating the XML manipulation code from the graphical interface into the basic game representation library.

- Implementing in C++ the reader/writer for Game Theory Explorer files (a first version of this is available in Python in the gambit.gte module).

- Unifying the **gambit-nfg2tex** and **gambit-nfg2html** command line tools into a **gambit-convert** tool, which would convert from and to many file formats.

- Extend all the Gambit command-line tools to read files of any accepted format, and not just .efg and .nfg files.

- **Languages**: C++; Python/Cython optional; XML experience helpful

- **Prerequisites**: Introductory game theory for familiarity with terminology of the objects in a game; undergraduate-level software engineering experience.

Structure equilibrium calculations using the strategy pattern

Gambit's architecture packages methods for finding Nash equilibria as standalone command-line tools. Owing to different histories in implementing these methods, internally the interfaces to these methods at the C++ level are quite heterogeneous. In some cases, something like the "strategy pattern" has been used to encapsulate these algorithms. In others, the interface is simply a global-scope function call with little or no structured interface.

This project would involve organizing all these interfaces in a consistent and unified way using the "strategy pattern." One can see an emerging structure in the **gambit-enumpure** implementation at `src/tools/enumpure/enumpure.cc` in the master branch of the git repository. The idea would be to develop a unified framework for being able to interchange methods to compute Nash equilibria (or other concepts) on games. If the project were to go well, as an extension these interfaces could then be wrapped in the Python API for direct access to the solvers (which are currently called via the command-line tools).

- **Languages**: C++.

- **Prerequisites**: Introductory game theory for familiarity with terminology of the objects in a game; undergraduate-level software engineering experience.

Implement Strategic Restriction of a game in C++

Gambit has a concept of a `StrategySupport` (defined in `src/libgambit/stratspt.h` and `src/libgambit/stratspt.cc`), which is used, among other things, to represent a game where strictly dominated strategies have been eliminated (which can be useful in improving the efficiency of equilibrium computations). The implementation of this has historically been awkward. Proper OO design would suggest that a `StrategySupport` should be able to be used anywhere a `Game` could be used, but this is not the case. In practice, a `StrategySupport` has just been a subset of the strategies of a game, which has to be passed along as a sidecar to the game in order to get anything done.

Recently, in the Python API, the model for dealing with this has been improved. In Python, there is a `StrategicRestriction` of a game, which in fact can be used seamlessly anyplace a game can be used. Separately, there is a `StrategySupportProfile`, which is basically just a subset of strategies of a game. This separation of concepts has proven to be clean and useful.

The project would be to develop the concept of a strategic restriction in C++, using the Python API as a model, with the idea of ultimately replacing the `StrategySupport`.

- **Languages**: C++; Python/Cython useful for understanding the current implementation in Python.

- **Prerequisites**: Introductory game theory for familiarity with terminology of the objects in a game; undergraduate-level software engineering experience.

Implement Behavior Restriction of a game in Python

The Python API has a concept of a `StrategicRestriction` of a game, which is the restriction of a game to a subset of strategies for each player. This restriction can be used seamlessly anywhere a game can be used.

This project would develop a parallel concept of a `BehaviorRestriction`. Logically this is similar to a `StrategicRestriction`, except that instead of operating on the set of reduced strategic game strategies, it would operate on behavior strategies (actions at information sets) in a game tree.

This is a bit more challenging than the `StrategicRestriction` because of the need to be able to traverse the resulting game tree. Removing actions from a game can result in entire subtrees of the game being removed, which can then include the removal of information sets from the game so restricted.

The idea of this project is to carry out the implementation in Python/Cython first, as the experience from the strategic restriction project was that the more rapid prototyping possible in Python was a big help. However, as the ultimate goal will be to provide this at the C++ level, there is also the possibility of attacking the problem directly in C++ as well.

- **Langauges**: Python/Cython; C++.

- **Prerequisites**: Introductory game theory for familiarity with terminology of the objects in a game; undergraduate-level software engineering experience.

Implementing algorithms for finding equilibria in games

Each of the following are separate ideas for open projects on computing equilibria and other interesting quantities on games. Each of these is a single project For GSoC applications, you should select exactly one of these, as each is easily a full summer's worth of work (no matter how easy some of them may seem at first read!)

Enumerating all equilibria of a two-player bimatrix game using the EEE algorithm

The task is to implement the EEE algorithm, which is a published algorithm to enumerate all extreme equilibria of a bimatrix game.

- **Languages:** C, Java

- **Prerequisites:** Background in game theory, basic linear algebra and linear programming. Experience with programs of at least medium complexity so that existing code can be expanded.

Fuller details:

The task is to implement the EEE algorithm, which is a published algorithm to enumerate all extreme equilibria of a bimatrix game.

The most up-to-date version can be found in Sections 7 and 8 of

> D. Avis, G. Rosenberg, R. Savani, and B. von Stengel (2010), Enumeration of Nash equilibria for two-player games. Economic Theory 42, 9-37.

> http://www.maths.lse.ac.uk/Personal/stengel/ETissue/ARSvS.pdf

Extra information, including some code, is provided in the following report:

> G. Rosenberg (2004), Enumeration of All Extreme Equilibria of Bimatrix Games with Integer Pivoting and Improved Degeneracy Check. CDAM Research Report LSE-CDAM-2004-18.

> http://www.cdam.lse.ac.uk/Reports/Files/cdam-2005-18.pdf

The original algorithm was described in the following paper:

C. Audet, P. Hansen, B. Jaumard, and G. Savard (2001), Enumeration of all extreme equilibria of bimatrix games. SIAM Journal on Scientific Computing 23, 323–338.

The implementation should include a feature to compare the algorithm's output (a list of extreme equilibria) with the ouput of other algorithms for the same task (e.g. `lrsnash`).

In addition a framework that compares running times (and the number of recursive calls, calls to pivoting methods, and other crucial operations) should be provided. The output should record and document the computational experiments so that they can be reproduced, in a general setup - sufficiently documented - that can be used for similar comparisons.

Improve integration and testing of Gametracer

Gambit incorporates the Gametracer package to provide implementations of two methods for computing equilibria, *gambit-gnm* and *gambit-ipa*. The integration is rather crude, as internally the program converts the game from native Gambit representation into Gametracer's representation, and the converts the output back. Using Gametracer's implementations as a starting point, refactor the implementation to use Gambit's native classes directly, and carry out experiments on the reliability and performance of the algorithms.

- **Languages:** C++
- **Prerequisites:** Some level of comfort with linear algebra; enjoyment of refactoring code.

Interface with lrslib

Gambit's *gambit-enummixed* tool computes all extreme Nash equilibria of a two-player game. There is another package, lrslib by David Avis, which implements the same algorithm more efficiently and robustly. There is a partial interface with an older version of lrslib in the Gambit source tree, which has proven not to be very reliable. The project is to complete the integration and testing of the lrslib integration.

- **Languages:** C/C++
- **Prerequisites:** Some level of comfort with linear algebra.

Finding equilibria reachable by Lemke's algorithm with varying "covering vectors"

Related to the Lemke-Howson method above, but with a slightly different algorithm that has an extra parameter, called the "covering vector". That parameter can serve a randomly selected starting point of the computation and potentially reach many more equilibria.

- **Prerequisites:** Theoretical understanding of the Lemke-Howson method or of the Simplex algorithm for Linear Programming. Literature exists that is accessible for students with at least senior-level background in computer science, mathematics or operations research. An existing implementation of a Lemke-Howson style pivoting algorithm should be adapted with suitable alterations.

Computing the index of an equilibrium component

The task is to implement a published algorithm to compute the so-called index of an equilibrium component in a bimatrix game. This component is the output to an existing enumeration algorithm.

- **Languages:** C
- **Prerequisites:** Senior-level mathematics, interest in game theory and some basic topology.

Fuller details:

The aim of this project is to implement an existing algorithm that finds the index of an equilibrium component. The relevant description of this is chapter 2 of

---

Anne Balthasar, Geometry and Equilibria in Bimatrix Games, PhD Thesis, London School of Economics, 2009.

http://www.maths.lse.ac.uk/Personal/stengel/phds/#anne

which are pages 21-41 of http://www.maths.lse.ac.uk/Personal/stengel/phds/anne-final.pdf

The mathematics in this chapter are pretty scary (in particular section 2.2, which is however not needed) but the final page 41 which describes the algorithm is less scary.

Nevertheless, this is rather advanced material because it builds on several different existing algorithms (for finding extreme equilibria in bimatrix games, and "cliques" that define convex sets of equilibria, and their non-disjoint unions that define "components"). It requires the understanding of what equilibria in bimatrix games are about. These algorithms are described in

D. Avis, G. Rosenberg, R. Savani, and B. von Stengel (2010), Enumeration of Nash equilibria for two-player games. Economic Theory 42, 9-37.

http://www.maths.lse.ac.uk/Personal/stengel/ETissue/ARSvS.pdf

and students who do not eventually understand that text should not work on this project. For this reason, at least senior-level (= third year) mathematics is required in terms of mathematical maturity. In the Avis et al. (2010) paper, pages 19-21 describe the lexicographic method for pivoting as it is used in the simplex method for linear programming. A variant of this lexicographic method is used in the chapter by Anne Balthasar. Understanding this is a requirement to work on this project (and a good test of how accessible all this is).

We give here two brief examples that supplement the above literature. Consider the following bimatrix game. It is very simple, and students of game theory may find it useful to first find out on their own what the equilibria of this game are:

```
2 x 2 Payoff matrix A:

1  1
0  1

2 x 2 Payoff matrix B:

1  1
0  1

EE = Extreme Equilibrium, EP = Expected Payoff

EE  1  P1:  (1)  1  0  EP=  1  P2:  (1)  1  0  EP=  1
EE  2  P1:  (1)  1  0  EP=  1  P2:  (2)  0  1  EP=  1
EE  3  P1:  (2)  0  1  EP=  1  P2:  (2)  0  1  EP=  1

Connected component 1:
{1, 2}  x  {2}
{1}  x  {1, 2}
```

This shows the following: there are 3 Nash equilibria, which partly use the same strategies of the two players, which are numbered (1), (2) for each player. It will take a bit of time to understand the above output. For our purposes, the bottom "component" is most relevant: It has two lines, and {1, 2} x {2} means that equilibrium (1),(2) - which is according to the previous list the strategy pair (1,0), (1,0) as well as (2),(2), which is (0,1), (1,0) are "extreme equilibria", and moreover any convex combination of (1) and (2) of player 1 - this is the first {1, 2} - can be combined with strategy (2) of player 2. This is part of the "clique" output of Algorithm 2 on page 19 of Avis et al. (2010). There is a second such convex set of equilibria in the second line, indicated by {1} x {1, 2}. Moreover, these two convex sets intersect (in the equilibrium (1),(2)) and form therefore a "component" of equilibria. For such a component, the index has to be found, which happens to be the integer 1 in this case.

The following bimatrix game has also two convex sets of Nash equilibria, but they are disjoint and therefore listed as separate components on their own:

```
3 x 2 Payoff matrix A:

1   1
0   1
1   0

3 x 2 Payoff matrix B:

2   1
0   1
0   1

EE = Extreme Equilibrium, EP = Expected Payoff

Rational Output

EE  1  P1:  (1)    1    0    0  EP=  1  P2:  (1)  1  0   EP= 2
EE  2  P1:  (2)  1/2  1/2    0  EP=  1  P2:  (2)  0  1   EP= 1
EE  3  P1:  (3)  1/2    0  1/2  EP=  1  P2:  (1)  1  0   EP= 1
EE  4  P1:  (4)    0    1    0  EP=  1  P2:  (2)  0  1   EP= 1

Connected component 1:
{1, 3}   x   {1}

Connected component 2:
{2, 4}   x   {2}
```

Here the first component has index 1 and the second has index 0. One reason for the latter is that if the game is slightly perturbed, for example by giving a slightly lower payoff than 1 in row 2 of the game, then the second strategy of player 1 is strictly dominated and the equilibria (2) and (4) of player 1, and thus the entire component 2, disappear altogether. This can only happen if the index is zero, so the index gives some useful information as to whether an equilibrium component is "robust" or "stable" when payoffs are slightly perturbed.

Enumerating all equilibria of a two-player game tree

Extension of an existing algorithm for enumerating all equilibria of a bimatrix game to game trees with imperfect information using the so-called "sequence form". The method is described in abstract form but not implemented.

- **Langauges:** C++

- **Prerequisites:** Background in game theory and basic linear algebra. Experience with programs of at least medium complexity so that existing code can be expanded.

Solving for equilibria using polynomial systems of equations

The set of Nash equilibrium conditions can be expressed as a system of polynomial equations and inequalities. The field of algebraic geometry has been developing packages to compute all solutions to a system of polynomial equations. Two such packages are PHCpack and Bertini. Gambit has an experimental interface, written in Python, to build the required systems of equations, call out to the solvers, and identify solutions corresponding to Nash equilibria. Refactor the implementation to be more flexible and Pythonic, and carry out experiments on the reliability and performance of the algorithms.

- **Languages:** Python

- **Prerequisites:** Experience with text processing to pass data to and from the external solvers.

Implement Herings-Peeters homotopy algorithm to compute Nash equilibria

Herings and Peeters (Economic Theory, 18(1), 159-185, 2001) have proposed a homotopy algorithm to compute Nash equilibria. They have created a first implementation of the method in Fortran, using hompack. Create a Gambit implementation of this method, and carry out experiments on the reliability and performance of the algorithms.

- **Languages:** C/C++, ability to at least read Fortran

- **Prerequisites:** Basic game theory and knowledge of pivoting algorithms like the Simplex method for Linear Programming or the Lemke-Howson method for games. Senior-level mathematics, mathematical economics, or operations research.

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.

General information

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

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 git://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; so, for example, `maint13` refers to the latest commit for Gambit version 13.x.x. Bug fixes should typically be based off of this branch.

Supported compilers

Currently, gcc is the only compiler supported. The version of gcc needs to be new enough to handle templates correctly. The oldest versions of gcc known to compile Gambit are 3.4.6 (Linux, Ubuntu) and 3.4.2 (MinGW for Windows, Debian stable).

If you wish to use another compiler, the most likely stumbling block is that Gambit uses templated member functions for classes, so the compiler must support these. (Version of gcc prior to 3.4 do not, for example.)

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, which can be gotten from http://www.mingw.org. We prefer MinGW over Cygwin because MinGW will create native Windows applications, whereas Cygwin requires an extra compatibility layer.

For OS X users

For building the command-line tools only, one should follow the instructions for Un*x/Linux platforms above. `make install` will install the command-line tools into `/usr/local/bin` (or the path specified in the `configure` step).

To build the graphical interface, wxWidgets 2.9.5 or higher is recommended, although 2.8.12 should also be suitable. (The interface will build with wxWidgets 2.9.4, but there is a bug in wxWidgets involving drag-and-drop which renders the graphical interface essentially unusable.)

Snow Leopard (OS X 10.8) users will have to take some extra steps to build wxWidgets if 2.8.12 is used. wxWidgets 2.8.12 requires the 10.6 SDK to build the using Cocoa; this has been removed by Apple in recent editions of XCode. Download and unpack the 10.6 SDK from an earlier XCode version into `/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.6.sdk`. With that in place, unpack the wxWidgets sources, and from the root directory of the wxWidgets sources, do:

```
mkdir build-debug
cd build-debug
arch_flags="-arch i386" CFLAGS="$arch_flags" CXXFLAGS="$arch_flags" \
```

```
   CPPFLAGS="$arch_flags" LDFLAGS="$arch_flags" OBJCFLAGS="$arch_flags" \
   OBJCXXFLAGS="$arch_flags" \
   ../configure  \
   --with-macosx-version-min=10.6 \
   --with-macosx-sdk=/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.
↪platform/Developer/SDKs/MacOSX10.6.sdk \
   --prefix="$(pwd)" --disable-shared --enable-debug --enable-unicode
make
```

Then, when configuring Gambit, use:

```
arch_flags="-arch i386" CFLAGS="$arch_flags" CXXFLAGS="$arch_flags" \
   CPPFLAGS="$arch_flags" LDFLAGS="$arch_flags" OBJCFLAGS="$arch_flags" \
   OBJCXXFLAGS="$arch_flags" \
   ./configure --with-wxdir=WXPATH/build-debug
make osx-bundle
```

where `WXPATH` is the path at which you have the wxWidgets sources unpacked. These steps are not required for wxWidgets 2.9.5 or higher.

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.

The graphical interface and wxWidgets

Gambit requires wxWidgets version 2.8.0 or higher for the graphical interface, although 2.9.5 or higher is recommended. 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.

Building the Python extension

The *Python extension for Gambit* is in src/python in the Gambit source tree. Prerequisite packages include setuptools, Cython, IPython, and scipy.

Building the extension follows the standard approach:

```
cd src/python
python setup.py build
sudo python setup.py install
```

There is a set of test cases in src/python/gambit/tests. These can be exercised via nosetests (requires Python package nose):

```
cd src/python/gambit/tests
nosetests
```

Once installed, simply `import gambit` 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.

Conventions common to all file formats

Several conventions are common to the interpretation of the file formats listed below.

Whitespace is not significant. In general, whitespace (carriage returns, horizontal and vertical tabs, and spaces) do not have an effect on the meaning of the file. The only exception is inside explicit double-quotes, where all characters are significant. The formatting shown here is the same as generated by the Gambit code and has been chosen for its readability; other formattings are possible (and legal).

Text labels. Most objects in an extensive game may be given textual labels. These are prominently used in the graphical interface, for example, and it is encouraged for users to assign nonempty text labels to objects if the game is going to be viewed in the graphical interface. In all cases, these labels are surrounded by the quotation character (''). The use of an explicit '' character within a text label can be accomplished by preceding the embedded '' characters with a backwards slash (). **Example 5-1. Escaping quotes in a text label**

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. yNumerical data, namely, the payoffs at outcomes, and the action probabilities for chance nodes, may be expressed in integer, decimal, or rational formats. In all cases, numbers are understood by Gambit to be exact, and represented as such internally. For example, the numerical entries 0.1 and 1/10 represent the same quantity.

In versions 0.97 and prior, Gambit distinguished between floating point and rational data. In these versions, the quantity 0.1 was represented interally as a floating-point number. In this case, since 0.1 does not have an exact representation in binary floating point, the values 0.1 and 1/10 were not identical, and some methods for computing equilibria could give (slightly) different results for games using one versus the other. In particular, using rational-precision methods on games with the floating point numbers could give unexpected output, since the conversion of 0.1 first to floating-point then to rational would involve roundoff error. This is largely of technical concern, and the current Gambit implementation now behaves in such a way as to give the "expected" result when decimal numbers appear in the file format.

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 interepreted as incremental payoffs; the payoff to a player for a given path through the tree is interpreted as the sum of the payoffs at the outcomes encountered on that path (including at the terminal node). This is ideal for the representation of games with well- defined"stages"; see, for example, the file bayes2a.efg in the Gambit distribution for a two-stage example of the Bayesian game represented previously.

In the following lists, fields which are omittable according to the above rules are indicated by the label (optional).

Format of chance (nature) nodes. Entries for chance nodes begin with the character c . Following this, in order, are

- a text string, giving the name of the node

- a positive integer specifying the information set number

- (optional) the name of the information set

- (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.

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 prologus. 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.

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.

The action graph game (.agg) file format

Action graph games (AGGs) are a compact representation of simultaneous-move games with structured utility functions. For more information on AGGs, the following paper gives a comprehensive discussion.

> A.X. Jiang, K. Leyton-Brown and N. Bhat, Action-Graph Games, Games and Economic Behavior, Volume 71, Issue 1, January 2011, Pages 141-173.

Each file in this format describes an action graph game. In order for the file to be recognized as AGG by GAMBIT, the initial line of the file should be:

```
#AGG
```

The rest of the file consists of 8 sections, separated by whitespaces. Lines with starting '#' are treated as comments and are allowed between sections.

1. The number of players, n.

2. The number of action nodes, |S|.

3. The number of function nodes, |P|.

4. Size of action set for each player. This is a row of n integers:

   $|S_1|\ |S_2|\ ....\ |S_n|$

5. Each Player's action set. We have N rows; row i has $|S_i|$ integers in ascending order, which are indices of Action nodes. Action nodes are indexed from 0 to $|S|-1$.

6. The Action Graph. We have |S|+|P| nodes, indexed from 0 to |S|+|P|-1. The function nodes are indexed after the action nodes. The graph is represented as (|S|+|P|) neighbor lists, one list per row. Rows 0 to |S|-1 are for action nodes; rows |S| to |S|+|P|-1 are for function nodes. In each row, the first number |v| specifies the number of neighbors of the node. Then follows |v| numbers, corresponding to the indices of the neighbors.

   We require that each function node has at least one neighbor, and the neighbors of function nodes are action nodes. The action graph restricted to the function nodes has to be a directed acyclic graph (DAG).

7. Signatures of functions. This is |P| rows, each specifying the mapping f_p that maps from the configuration of the function node p's neighbors to an integer for p's "action count". Each function is specified by its "signature" consisting of an integer type, possibly followed by further parameters. Several types of mapping are implemented:

   - Types 0-3 require no further input.

     - Type 0: Sum. i.e. The action count of a function node p is the sum of the action counts of p's neighbors.

     - Type 1: Existence: boolean for whether the sum of the counts of neighbors are positive.

     - Type 2: The index of the neighbor with the highest index that has non-zero counts, or |S|+|P| if none applies.

     - Type 3: The index of the neighbor with the lowest index that has non-zero counts, or |S|+|P| if none applies.

   - Types 10-13 are extended versions of type 0-3, each requiring further parameters of an integer default value and a list of weights, |S| integers enclosed in square brackets. Each action node is thus associated with an integer weight.

     - Type 10: Extended Sum. Each instance of an action in p's neighborhood being chosen contributes the weight of that action to the sum. These are added to the default value.

     - Type 11: Extended Existence: boolean for whether the extended sum is positive. The input default value and weights are required to be nonnegative.

     - Type 12: The weight of the neighbor with the highest index that has non-zero counts, or the default value if none applies.

     - Type 13: The weight of the neighbor with the lowest index that has non-zero counts, or the default value if none applies.

   The following is an example of the signatures for an AGG with three action nodes and two function nodes:

```
2
10 0 [2 3 4]
```

8. The payoff function for each action node. So we have |S| subblocks of numbers. Payoff function for action s is a mapping from configurations to real numbers. Configurations are represented as a tuple of integers; the size of the tuple is the size of the neighborhood of s. Each configuration specifies the action counts for the neighbors of s, in the same order as the neighbor list of s.

   The first number of each subblock specifies the type of the payoff function. There are multiple ways of representing payoff functions; we (or other people) can extend the file format by defining new types of payoff functions. We define two basic types:

   Type 0 The complete representation. The set of possible configurations can be derived from the action graph. This set of configurations can also be sorted in lexicographical order. So we can just specify the payoffs without explicitly giving the configurations. So we just need to give one row of real numbers, which correspond to payoffs for the ordered set of configurations.

If action s is in multiple players' action sets (say players i, j), then it is possible that the set of possible configurations given s_i is different from the set of possible configurations given s_j. In such cases, we need to specify payoffs for the union of the sets of configurations (sorted in lexicographical order).

Type 1 The mapping representation, in which we specify the configurations and the corresponding payoffs. For the payoff function of action s, first give Delta_s, the number of elements in the mapping. Then follows Delta_s rows. In each row, first specify the configuration, which is a tuple of integers, enclosed by a pair of brackets "[" and "]", then the payoff. For example, the following specifies a payoff function of type 1, with two configurations:

```
1 2
[1 0] 2.5
[1 1] -1.2
```

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

Articles on computation of Nash equilibria

General game theory articles and texts

Textbooks and general reference

Detailed table of contents

Or, see a *more detailed table of contents*.

Bibliography

[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.

[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.

Python Module Index

g