.. _pygambit-user:

User guide

Example: One-shot trust game with binary actions
~~~~~~

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

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

We create a game with an extensive representation using :py:meth:‘.Game.new_tree‘:

.. ipython:: python

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

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

.. ipython:: python

```
g.root
g.root.children
```

To extend a game from an existing terminal node, use :py:meth:‘.Game.append_move‘:

.. ipython:: python

```
g.append_move(g.root, "Buyer", ["Trust", "Not trust"])
g.root.children
```

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

.. ipython:: python

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

Now that we have the moves of the game defined, we add payoffs. Payoffs are associated with
an :py:class:‘.Outcome‘; each :py:class:‘Outcome‘ has a vector of payoffs, one for each player,
and optionally an identifying text label. First we add the outcome associated with the
~~~~~~

Seller proving themselves trustworthy:

.. ipython:: python

   g.set_outcome(g.root.children[0].children[0], g.add_outcome([1, 1], label="Trustworthy"))

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

.. ipython:: python

   g.set_outcome(g.root.children[0].children[1], g.add_outcome([-1, 2], label="Untrustworthy"))

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

.. ipython:: python

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

Nodes without an outcome attached are assumed to have payoffs of zero for all players.
Therefore, adding the outcome to this latter terminal node is not strictly necessary in Gambit,
but it is useful to be explicit for readability.

.. [Kre90] Kreps, D. (1990) "Corporate Culture and Economic Theory."
   In J. Alt and K. Shepsle, eds., *Perspectives on Positive Political Economy*,
   Cambridge University Press.

.. _pygambit.user.poker:

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

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

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

We can build this game using the following script::

```
g = gbt.Game.new_tree(players=["Alice", "Bob"],
                      title="One card poker game, after Myerson (1991)")
```
~~~~~~

```
    g.append_move(g.root, g.players.chance, ["King", "Queen"])
    for node in g.root.children:
       g.append_move(node, "Alice", ["Raise", "Fold"])
    g.append_move(g.root.children[0].children[0], "Bob", ["Meet", "Pass"])
    g.append_infoset(g.root.children[1].children[0],
                     g.root.children[0].children[0].infoset)
    alice_winsbig = g.add_outcome([2, -2], label="Alice wins big")
    alice_wins = g.add_outcome([1, -1], label="Alice wins")
    bob_winsbig = g.add_outcome([-2, 2], label="Bob wins big")
    bob_wins = g.add_outcome([-1, 1], label="Bob wins")
    g.set_outcome(g.root.children[0].children[0].children[0], alice_winsbig)
    g.set_outcome(g.root.children[0].children[0].children[1], alice_wins)
    g.set_outcome(g.root.children[0].children[1], bob_wins)
    g.set_outcome(g.root.children[1].children[0].children[0], bob_winsbig)
    g.set_outcome(g.root.children[1].children[0].children[1], alice_wins)
    g.set_outcome(g.root.children[1].children[1], bob_wins)
```

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

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

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

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

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

The call :py:meth:‘.Game.append_infoset‘ adds a move at a terminal node as part of an existing information set (represented in ‘‘pygambit‘‘ as an :py:class:‘.Infoset‘).

.. [Mye91] Myerson, Roger B. (1991) *Game Theory: Analysis of Conflict*.
Cambridge: Harvard University Press.

.. [RUW08] Reiley, David H., Michael B. Urbancic and Mark Walker. (2008)
"Stripped-down poker: A classroom game with signaling and bluffing."
The Journal of Economic Education 39(4): 323-341.

Building a strategic game

Games in strategic form, also referred to as normal form, are represented solely by a collection of payoff tables, one per player. The most direct way to create a strategic game is via :py:meth:‘.Game.from_arrays‘. This function takes one

n-dimensional array per player, where n is the number of players in the game.
The arrays can be any object that can be indexed like an n-times-nested Python list;
so, for example, ‘numpy‘ arrays can be used directly.

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

```
.. ipython:: python

   import numpy as np
   m = np.array([[8, 2], [10, 5]])
   g = gbt.Game.from_arrays(m, np.transpose(m))
   g
```

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

```
.. _pygambit.user.numbers:
```

Representation of numerical data of a game
~~~~~~

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

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

```
.. ipython:: python

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

The default when creating a new move for chance is that all actions are chosen with
equal probability. These probabilities are represented as rational numbers,
using ‘‘pygambit‘‘’s :py:class:‘.Rational‘ class, which is derived from Python’s
‘fractions.Fraction‘. Numerical data can be set as rational numbers:

```
.. ipython:: python

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

They can also be explicitly specified as decimal numbers:

```
.. ipython:: python

   g.set_chance_probs(g.root.infoset,
```
~~~~~~

```
                [gbt.Decimal(".25"), gbt.Decimal(".50"), gbt.Decimal(".25")])
  [act.prob for act in g.root.infoset.actions]
```

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

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

```
.. ipython:: python

  g.set_chance_probs(g.root.infoset, ["1/4", "1/2", "1/4"])
  [act.prob for act in g.root.infoset.actions]

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

As a further convenience, ‘‘pygambit‘‘ will accept Python ‘‘int‘‘ and ‘‘float‘‘ values. ‘‘int‘‘ values are always interpreted as :py:class:‘.Rational‘ values. ‘‘pygambit‘‘ attempts to render ‘float‘ values in an appropriate :py:class:‘.Decimal‘ equivalent. In the majority of cases, this creates no problems. For example,

```
.. ipython:: python

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

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

```
.. ipython:: python
  :okexcept:

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

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

```
.. ipython:: python

  1/3 + 1/3 + 1/3
```

In checking whether these probabilities sum to one, ‘‘pygambit‘‘ first converts each of the probabilitiesto a :py:class:‘.Decimal‘ representation, via the following method

```
.. ipython:: python

  gbt.Decimal(str(1/3))
```

and the sum-to-one check then fails because

.. ipython:: python

   gbt.Decimal(str(1/3)) + gbt.Decimal(str(1/3)) + gbt.Decimal(str(1/3))

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

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

Reading a game from a file
~~~~~~~~~~~~~~~~~~~~~~~~~~

Games stored in existing Gambit savefiles can be loaded using :meth:‘.Game.read_game‘:

.. ipython:: python
   :suppress:

   cd ../contrib/games

.. ipython:: python

   g = gbt.Game.read_game("e02.nfg")
   g

.. ipython:: python
   :suppress:

   cd ../../doc

Computing Nash equilibria
~~~~~~~~~~~~~~~~~~~~~~~~~

Interfaces to algorithms for computing Nash equilibria are provided in :py:mod:‘pygambit.nash‘.

Method	Python function
:ref:‘gambit-enumpure <gambit-enumpure>‘	:py:func:‘pygambit.nash.enumpure_solve‘
:ref:‘gambit-enummixed <gambit-enummixed>‘	:py:func:‘pygambit.nash.enummixed_solve‘
:ref:‘gambit-lp <gambit-lp>‘	:py:func:‘pygambit.nash.lp_solve‘
:ref:‘gambit-lcp <gambit-lcp>‘	:py:func:‘pygambit.nash.lcp_solve‘
:ref:‘gambit-liap <gambit-liap>‘	:py:func:‘pygambit.nash.liap_solve‘
:ref:‘gambit-logit <gambit-logit>‘	:py:func:‘pygambit.nash.logit_solve‘

:ref:‘gambit-simpdiv <gambit-simpdiv>‘ :py:func:‘pygambit.nash.simpdiv_solve‘
:ref:‘gambit-ipa <gambit-ipa>‘ :py:func:‘pygambit.nash.ipa_solve‘
:ref:‘gambit-gnm <gambit-gnm>‘ :py:func:‘pygambit.nash.gnm_solve‘
==========
==========

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

For two-player games, :py:func:‘.lcp_solve‘ can compute Nash equilibria directly using the extensive representation. Assuming that ‘‘g‘‘ refers to the game

```
.. ipython:: python
   :suppress:

   g = gbt.Game.read_game("poker.efg")
```

```
.. ipython:: python

   result = gbt.nash.lcp_solve(g)
   result
   len(result.equilibria)
```

The result of the calculation is returned as a :py:class:‘.NashComputationResult‘ object.
The set of equilibria found is reported in :py:attr:‘.NashComputationResult.equilibria‘;
in this case, this is a list of mixed behavior profiles.
A mixed behavior profile specifies, for each information set, the probability distribution over actions at that information set.
Indexing a :py:class:‘.MixedBehaviorProfile‘ by a player gives a :py:class:‘.MixedBehavior‘, which specifies probability distributions at each of the player’s information sets:

```
.. ipython:: python

   eqm = result.equilibria[0]
   eqm["Alice"]
```

In this case, at Alice’s first information set, the one at which she has the King, she always raises.
At her second information set, where she has the Queen, she sometimes bluffs, raising with probability one-third.
The probability distribution at an information set is represented by a :py:class:‘.MixedAction‘.
:py:meth:‘.MixedBehavior.mixed_actions‘ iterates over these for the player:

```
.. ipython:: python

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

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

```
.. ipython:: python

   {infoset: mixed_action["Raise"] for infoset, mixed_action in eqm["Alice"].mixed_actions()}
```

In larger games, labels may not always be the most convenient way to refer to specific actions. We can also index profiles directly with :py:class:‘.Action‘ objects.
So an alternative way to extract the probabilities of playing "Raise" would be by iterating Alice’s list of actions:

.. ipython:: python

   {action.infoset: eqm[action] for action in g.players["Alice"].actions if action.label == "Raise"}

Looking at Bob’s strategy,

.. ipython:: python

   eqm["Bob"]

Bob meets Alice’s raise two-thirds of the time. The label "Raise" is used in more than one information set for Alice, so in the above we had to specify information sets when indexing.
When there is no ambiguity, we can specify action labels directly. So for example, because Bob has only one action named "Meet" in the game, we can extract the probability that Bob plays "Meet" by:

.. ipython:: python

   eqm["Bob"]["Meet"]

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

.. ipython:: python

   eqm["Meet"]

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

.. ipython:: python

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

Bob’s indifference between his actions arises because of his beliefs given Alice’s strategy.
:py:meth:‘.MixedBehaviorProfile.belief‘ returns the probability of reaching a node, conditional on its information set being reached:

.. ipython:: python

   {node: eqm.belief(node) for node in g.players["Bob"].infosets[0].members}

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

.. ipython:: python

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

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

.. ipython:: python

   {node: eqm.node_value("Bob", node) for node in g.players["Bob"].infosets[0].members}

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

.. ipython:: python

   eqm.payoff("Alice")
   eqm.payoff("Bob")

The equilibrium computed expresses probabilities in rational numbers. Because the numerical data of games in Gambit :ref:‘are represented exactly <pygambit.user.numbers>‘, methods which are specialized to two-player games, :py:func:‘.lp_solve‘, :py:func:‘.lcp_solve‘, and :py:func:‘.enummixed_solve‘, can report exact probabilities for equilibrium strategy profiles. This is enabled by default for these methods.

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

.. ipython:: python

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

In the strategic form of this game, Alice has four strategies. The generated strategy labels list the action numbers taken at each information set. We can therefore apply a method which operates on a strategic game to any game with an extensive representation

.. ipython:: python

   result = gbt.nash.gnm_solve(g)
   result

:py:func:‘.gnm_solve‘ can be applied to any game with any number of players, and uses a path-following process in floating-point arithmetic, so it returns profiles with probabilities expressed as floating-point numbers. This method operates on the strategic representation of the game, so the returned results are of type :py:class:‘~pygambit.gambit.MixedStrategyProfile‘, and specify, for each player, a probability distribution over that player’s strategies. Indexing a :py:class:‘.MixedStrategyProfile‘ by a player gives the probability distribution over that player’s strategies only.

.. ipython:: python

   eqm = result.equilibria[0]
   eqm["Alice"]

```
eqm["Bob"]
```

The expected payoff to a strategy is provided by :py:meth:‘.MixedStrategyProfile.strategy_value‘:

.. ipython:: python

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

The overall expected payoff to a player is returned by :py:meth:‘.MixedStrategyProfile.payoff‘:

.. ipython:: python

```
eqm.payoff("Alice")
eqm.payoff("Bob")
```

When a game has an extensive representation, we can convert freely between
:py:class:‘~pygambit.gambit.MixedStrategyProfile‘ and the corresponding
:py:class:‘~pygambit.gambit.MixedBehaviorProfile‘ representation of the same strategies
using :py:meth:‘.MixedStrategyProfile.as_behavior‘ and :py:meth:‘.MixedBehaviorProfile.as_strategy‘.

.. ipython:: python

```
eqm.as_behavior()
eqm.as_behavior().as_strategy()
```

Estimating quantal response equilibria
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

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

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

.. ipython:: python

```
g = gbt.Game.from_arrays(
    [[1.1141, 0], [0, 0.2785]],
    [[0, 1.1141], [1.1141, 0]],
    title="Ochs (1995) asymmetric matching pennies as transformed in McKelvey-Palfrey (1995)"
)
data = g.mixed_strategy_profile([[128*0.527, 128*(1-0.527)], [128*0.366, 128*(1-0.366)]])
```

Estimation of QRE is done using :py:func:‘.fit_fixedpoint‘.

.. ipython:: python

```
fit = gbt.qre.fit_fixedpoint(data)
```

The returned :py:class:‘.LogitQREMixedStrategyFitResult‘ object contains the results of the
estimation.
The results replicate those reported in [McKPal95]_, including the estimated value of lambda,
the QRE profile probabilities, and the log-likelihood.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Because ‘data‘ contains the empirical counts of play, and not just frequencies, the resulting log-likelihood is correct for use in likelihoood-ratio tests. [#f1]_

.. ipython:: python

```
print(fit.lam)
print(fit.profile)
print(fit.log_like)
```

.. rubric:: Footnotes

.. [#f1] The log-likelihoods quoted in [McKPal95]_ are exactly a factor of 10 larger than those obtained by replicating the calculation.