

USING COMPUTER ALGEBRA TO FIND NASH EQUILIBRIA

RUCHIRA S. DATTA

1. INTRODUCTION

Since von Neumann and Morgenstern's seminal *Theory of Games and Economic Behavior* was first published (1944), game theory has been widely used to model strategic interaction between rational agents. In the absence of binding contracts restraining them, the behavior of such agents in a particular game is predicted to be a *Nash equilibrium*. Defined by Nash (1951), Nash equilibria of finite games (i.e., with finite numbers of players, each with a finite number of pure strategies) can be characterized as solutions to systems of multilinear equalities and inequalities. The main tool for computing such equilibria today is the free software package Gambit of McKelvey, McLennan, and Turocy. However, techniques for solving systems of polynomial equations have continued evolving for several years since their implementation in this package. Here we experiment with the use of various general-purpose polynomial system solvers to solve polynomial equations arising from games. Our goal is to determine which of the algebraic techniques today performs best for these problems.

2. PRELIMINARIES

We restrict our attention to finite, noncooperative games (in which players are not constrained by binding contracts to remain in agreed-upon coalitions) in normal form, with more than two players. Such a game is specified as follows. There are N players, $1, \dots, N$. Player i has $d_i + 1$ pure strategies, $\sigma_{i0}, \dots, \sigma_{id_i}$. These are actions player i can take. For example, player 1, Alice, could have two pure strategies: σ_{10} “invest company profits in Amgen stock” or σ_{11} “use company profits for capital improvements”. The actions player i can take may be completely different from the actions player j can take. For example, the possible actions of player 2, Bob, might be σ_{20} “underwrite Amgen bonds” or σ_{21} “underwrite bonds for Alice's company”, and the possible actions of player 3, Chris, might be σ_{30} “buy Amgen stock” and σ_{31} “buy Alice's stock”. A pure strategy profile is a choice by each player of

a pure strategy, that is, an N -tuple $(\sigma_{1j_1}, \sigma_{2j_2}, \dots, \sigma_{Nj_N})$. The game specification is completed by giving for each such N -tuple and each player i , the payoff (or utility) $u_i(\sigma_{1j_1}, \sigma_{2j_2}, \dots, \sigma_{Nj_N})$ which will be accrued by player i when these choices are made. Any player may choose a convex combination of their pure strategies, called a *mixed strategy*. So a mixed strategy for player i is a $(d_i + 1)$ -tuple of positive numbers $(p_{i0}, \dots, p_{id_i})$ summing to 1. The payoffs accruing to the players in such a situation are given by multilinearity. Mixed strategies can be interpreted probabilistically, as indicating that player i will roll a die and take action σ_{i0} with probability p_{i0} , action σ_{i1} with probability p_{i1} , and so forth. In some cases it can also be interpreted in other ways, for example, “invest a fraction p_{i0} of the company profits in Amgen stock” and “use a fraction p_{i1} of the company profits for capital improvements.” The game is said to be in normal form since all players make a single move simultaneously, and in ignorance of each others’ moves.

Thus, to specify the game requires $N (d_1 + 1) \times (d_2 + 1) \times \dots \times (d_N + 1)$ tables of numbers, the payoff tables for each player. So the example of Alice and Bob would be completely specified by 3 $2 \times 2 \times 2$ tables. Alice’s table might be

$$(1) \quad (u_1(\sigma_{1i} \sigma_{2j} \sigma_{30})) = \begin{matrix} & \sigma_{20} & \sigma_{21} \\ \sigma_{10} & \begin{pmatrix} 3 & 2 \end{pmatrix} \\ \sigma_{11} & \begin{pmatrix} 1 & 5 \end{pmatrix} \end{matrix},$$

and

$$(2) \quad (u_1(\sigma_{1i} \sigma_{2j} \sigma_{31})) = \begin{matrix} & \sigma_{20} & \sigma_{21} \\ \sigma_{10} & \begin{pmatrix} 2 & 4 \end{pmatrix} \\ \sigma_{11} & \begin{pmatrix} 0 & 7 \end{pmatrix} \end{matrix}.$$

Bob’s table might be

$$(3) \quad (u_2(\sigma_{1i} \sigma_{2j} \sigma_{30})) = \begin{matrix} & \sigma_{20} & \sigma_{21} \\ \sigma_{10} & \begin{pmatrix} 5 & 4 \end{pmatrix} \\ \sigma_{11} & \begin{pmatrix} 3 & 2 \end{pmatrix} \end{matrix},$$

and

$$(4) \quad (u_2(\sigma_{1i} \sigma_{2j} \sigma_{31})) = \begin{matrix} & \sigma_{20} & \sigma_{21} \\ \sigma_{10} & \begin{pmatrix} 4 & 6 \end{pmatrix} \\ \sigma_{11} & \begin{pmatrix} 2 & 7 \end{pmatrix} \end{matrix}.$$

Chris's table might be

$$(5) \quad (u_3(\sigma_{1i} \sigma_{2j} \sigma_{30})) = \begin{matrix} & \sigma_{20} & \sigma_{21} \\ \sigma_{10} & \begin{pmatrix} 8 & 1 \end{pmatrix} \\ \sigma_{11} & \begin{pmatrix} 2 & 4 \end{pmatrix} \end{matrix},$$

and

$$(6) \quad (u_3(\sigma_{1i} \sigma_{2j} \sigma_{31})) = \begin{matrix} & \sigma_{20} & \sigma_{21} \\ \sigma_{10} & \begin{pmatrix} 5 & 3 \end{pmatrix} \\ \sigma_{11} & \begin{pmatrix} 1 & 7 \end{pmatrix} \end{matrix}.$$

A strategy profile is a Nash equilibrium if for each player i , i could not attain a (strictly) better payoff by changing only i 's own (mixed) strategy, leaving all others' strategies fixed. If the game were to be repeated, no player would be motivated to adopt a different strategy unilaterally. In the example, the strategy profiles $(\sigma_{10}, \sigma_{20}, \sigma_{30})$ is a Nash equilibrium. It is better for any player to go with Amgen, given that the other two players are doing so. Similarly, $(\sigma_{11}, \sigma_{21})$ is also a Nash equilibrium.

A Nash equilibrium is *totally mixed* if for each player, every pure strategy of that player occurs with positive probability in the mixed strategy. A totally mixed Nash equilibrium can only occur if for any player i , keeping all the other players' strategies fixed, the payoffs to i from each of i 's pure strategies are equal. Otherwise, i could improve i 's own payoff by omitting (choosing with zero probability) those pure strategies leading to lesser payoffs. These conditions give a system of polynomial equations. The unknowns in this system are the proportions allocated by each player to their various pure strategies to form their mixed strategies. The sum of each player's own unknowns is 1. The solutions to this polynomial system are called "quasi-equilibria". Only the solutions which are real and lie in the positive orthant represent actual Nash equilibria.

In the example, Alice's payoff from executing pure strategy σ_{i0} is $3p_{20}p_{30} + 2p_{21}p_{30} + 2p_{20}p_{31} + 4p_{21}p_{31}$. (In the probabilistic interpretation this is her expected payoff, conditioned on the event that she chooses strategy σ_{i0} .) Her payoff from executing pure strategy σ_{i1} is $1p_{20}p_{30} + 5p_{21}p_{30} + 0p_{20}p_{31} + 7p_{21}p_{31}$. Substituting in that $p_{20} = 1 - p_{21}$ and $p_{30} = 1 - p_{31}$ and equating these two expressions, we obtain that $3 - p_{21} - p_{31} + 3p_{21}p_{31} = 1 + 4p_{21} - p_{31} + 3p_{21}p_{31}$, i.e., $2 - 5p_{21} = 0$. Similarly, the other two equations in the system are $1 - 3p_{31} - 3p_{11}p_{31} = 0$ and $3 - 2p_{11} - 5p_{21} + p_{11}p_{21} = 0$. This system has a unique solution at $p_{11} = 5/8$, $p_{21} = 2/5$, $p_{31} = 8/39$. Since each of these lies between 0

and 1, (so p_{10} , p_{20} , and p_{30} are also positive), this is in fact a totally mixed Nash equilibrium (the unique one).

We concentrate on the problem of computing all totally mixed Nash equilibria. Once we have a procedure to do this, it can be used as a subroutine to compute all Nash equilibria. For every subset of the set of all pure strategies of all players not containing all of any particular player's pure strategies, one derives a new normal form game in which that subset of pure strategies is unavailable, and finds the totally mixed Nash equilibria for the new game. Then one checks if these would still be Nash equilibria if the deleted pure strategies were available. If so, then these are partially mixed Nash equilibria of the original game, that is, they are equilibria in which the probabilities allocated to the pure strategies in the subset are zero. In the special case when for each player, the subset contains all but one of that player's strategies, the resulting point is trivially a Nash equilibrium in the new game, and potentially a pure strategy Nash equilibrium of the old game. This is in fact how Gambit computes all Nash equilibria of a game with more than two players.

We see that the problem of computing totally mixed equilibria reduces to that of solving a system of polynomial equations subject to some inequality constraints. As we showed in Datta (2002), the solution set can be (stably isomorphic to) any algebraic variety (i.e., to the solution set of any polynomial system). However, Harsanyi (1973) showed that for any generic set of payoffs, the number of Nash equilibria is finite. In his formulation "generic" meant "except on a set of measure zero". From an algebraic point of view, this implies that for any assignment of payoff values outside of an algebraic subset of positive codimension in the space of all such assignments, the solution set to the polynomial system is a zero-dimensional algebraic variety. This paper focuses on applying various techniques to compute the complete set of totally mixed Nash equilibria.

3. THE STATUS QUO: Gambit

Gambit, developed by McKelvey, McLennan, and Turocy, is currently the standard software package for computing Nash equilibria. Most of the code focuses on solving two-person games. (Indeed, a large proportion of the game theory literature itself focuses on two-person games. This may be because the two-person situation is already quite rich and interesting, but it also may be partially due to the current inability to solve even moderate examples of games with more than three players.) For normal-form games with more than two players, the program can

compute all equilibria with a routine called `PolEnumSolve`. It can also compute single equilibria (or multiple equilibria, one at a time, with no guarantee at any point that all have been found) with several other algorithms. However, since the algebraic techniques (including `PolEnumSolve`) which we are comparing here solve for all equilibria, we will only consider `PolEnumSolve`. By default, `PolEnumSolve` solves for all Nash equilibria by recursing over the possible subsets of used strategies as explained above. But we chose not to recurse for ease of comparison with the other methods, which we will use to compute all (and only) the totally mixed Nash equilibria.

`PolEnumSolve` works by spatial subdivision, a technique often used as well in computer-aided geometric design. The algorithm starts with a higher-dimensional cube that contains the entire strategy space. It uses Newton iteration to find a solution within this cube. If there is one, it checks whether this solution is within the strategy space; if not it discards it and starts again. When it finds a bona fide solution, it checks that it is the unique solution within this cube—in fact, within the sphere circumscribing the cube. Basically, if the system were linear it could have no other solutions at all (since the solution set is zero-dimensional). Gambit checks that the linear Taylor approximation is good enough (i.e., the nonlinear part of the system is small enough) within this sphere to guarantee that there is no other root inside. If the solution cannot be guaranteed to be unique, then the cube is subdivided and the process is repeated within the smaller cubes.

We generated various games with random entries to try to solve with Gambit. Unfortunately the current version of Gambit (version 0.97.0.3, “Legacy”), is extremely unstable and crashes with a segmentation fault on many of the simplest games. The only games which it was able to solve with any consistency were the smallest case of more than two players, namely three players each with two pure strategies. (Even here many segmentation faults occurred.) These took it from 60 to 160 ms to solve. (All these computations, and all others reported here, were done on the same machine: a Dell Latitude C840 2.0GHz Mobile Pentium 4 laptop with 1GB RAM running Linux kernel version 2.4.19.)

4. PURE ALGEBRA: GRÖBNER BASES

The set of solutions to a system of polynomial equations is called an *algebraic variety*. Conversely, consider the set of polynomials which vanish on a set of points. Any sum of two such polynomials will also vanish on the same set, and so will any product of such a polynomial

with any other polynomial. These two conditions mean that the set of polynomials vanishing on a set is an *ideal*, i.e., closed under addition and under multiplication by a polynomial.

A *generating set* for an ideal is a set of elements such that every other element is a sum of products of these elements with other polynomials. It so happens that every polynomial ideal has a finite generating set. Given such a generating set, one might try to determine whether a particular candidate polynomial lies in the ideal by dividing by the polynomial generators. However, in general, the remainder is not uniquely determined, and may not be zero even though the candidate polynomial lies in the ideal. This state of affairs is remedied by requiring that the generating set satisfy certain technical conditions (Buchberger's criterion). If it does, it qualifies as a *Gröbner basis*.

A Gröbner basis is defined with respect to a particular *term order*. There is a natural partial order on monomials, namely that induced by divisibility (with 1 being the least monomial). A *term order* extends this partial order to a total order, while respecting multiplication. More precisely, if $m_1 < m_2$ for a term order $<$, then $mm_1 < mm_2$ for any monomial m . Every polynomial ideal has a Gröbner basis with respect to every term order. If we specify that the Gröbner basis must be *reduced*, then it is unique for a given term order. (This means that no term in any element of the basis can be divisible by the leading term of another element of the basis.) Gröbner bases can be used to solve many of the fundamental problems of computational algebra.

Perhaps the most intuitive term order is the *lexicographic* one. One specifies an ordering of the variables. Then in comparing two monomials, one first compares the powers of the heaviest (greatest) variable. If they are unequal, this is decisive; otherwise one compares the powers of the next heaviest variable, and so on.

The reduced Gröbner basis with respect to the lexicographic term order provably has higher degrees than the Gröbner basis with respect to some other term orders, and so the computational complexity of many algorithms is worsened when using this order. However, this term order in particular supports solving 0-dimensional polynomial systems. It follows from elimination theory that the reduced Gröbner basis will be triangular. That is, it will consist of a polynomial in which only one variable occurs, one in which that and another variable occur, one in which those two and a third variable occur, and so on. The roots of the first polynomial can be found numerically. Then each of these values can be substituted into the second polynomial, making it a polynomial only in the second variable. Solving this numerically gives the values of the second variable, and so on.

Two popular software packages for Gröbner basis computations are Macaulay2 and Singular. Unfortunately Macaulay2 is not currently well set up for solving polynomial systems, although support is planned for the future. In our tests it took about 10 ms to find a Gröbner basis for the case of 3 players with 2 strategies each, and 2.64 seconds to find a Gröbner basis for the case of 4 players with 2 strategies each. On larger instances it exited with a segmentation fault. Of course, this does not include the time to actually use the Gröbner basis to find the roots, which would require exporting the problem to another numerical solving routine such as one from the Netlib repository or in Matlab.

Singular, on the other hand, did much better. It comes with a standard library `solve.lib` for complex symbolic-numeric polynomial solving. Using the main routine `solve` from this library, we were able to solve the case of 3 players with 2 strategies each in <1 ms, and with 3 strategies each in 1 ms. The case of 4 players with 2 strategies each was solved in 70 ms.

While the above results seem promising, they are eclipsed by the performance of the polyhedral homotopy continuation method, to which we turn next.

5. POLYHEDRAL HOMOTOPY CONTINUATION: PHC

The polynomial systems we want to solve are very *sparse*. That is, given the total degree of each equation, we don't see all the terms that there could be in an equation of that degree. Specifically, these monomials are multilinear. Moreover, in the equation associated with player i , none of the variables associated with i appear, and in each term of such an equation, only one of the variables associated with any other particular player j can appear at a time.

The number of solutions to such a sparse 0-dimensional polynomial system is generically far smaller than the Bézout number obtained by multiplying total degrees. The sparsity of the system can be described in terms of the exponent vectors occurring in the monomials which occur in each polynomial. For a single polynomial, the convex hull of these vectors forms a lattice polytope, called its *Newton polytope*. The Minkowski sum of two Newton polytopes is obtained by translating one of them by each of the vectors in the other and taking the convex hull of the result. (For simplicity we describe two, but the definitions hold for arbitrary numbers.) One can subdivide the Minkowski sum into smaller lattice polytopes by following the course of the various faces during the translation. Some members of this subdivision will be

pure translates of one of the original polytopes, but the rest will be mixed, having faces coming from both polytopes. The mixed volume of the system is the normalized (with respect to the integer lattice) volume of these mixed elements, and amazingly enough, it gives the Bernstein number, which is defined as the number of roots of a generic polynomial system with those Newton polytopes.

This number is only a function of the Newton polytopes. The *polyhedral homotopy continuation* method, introduced by Huber and Sturmfels (1995), takes advantage of this. In general, one uses homotopy continuation to solve a polynomial system by first starting with another system of the same multidegree whose roots are obvious (and all distinct—roots with multiplicity are generally troublesome for numerical solvers), and gradually perturbing (or “morphing”) the coefficients towards the system of interest. At each step, one finds all the roots of the intermediate system by iterating from the roots of the system in the previous step. In this way one traces out a path from each root of the starting system to each root of the system of interest (which is why this method is also called “path-following”). The main drawback previously was that many of these paths would not lead to roots, because the starting system was generic and *dense*, whereas polynomial systems arising in practice are usually sparse. Thus the number of paths would explode with the size of the problem. However, with polyhedral homotopy continuation, the starting system is also chosen to be sparse in the same way as the system of interest. So only those paths which can lead to actual roots are followed.

Verschelde’s software package PHC for polyhedral homotopy continuation is in continuous development yet is very stable. Furthermore, it is well-documented and very simple to use. We were able to solve the following cases:

- 3 players with 2 pure strategies each: 2 roots found in 20ms.
- 3 players with 3 pure strategies each: 10 roots found in 350ms.
- 3 players with 4 pure strategies each: 56 roots found in 13s280ms.
- 3 players with 5 pure strategies each: 342 roots found in 3m19s540ms.
- 3 players with 6 pure strategies each: 2252 roots found in 48m41s870ms.
- 4 players with 2 pure strategies each: 9 roots found in 260ms.
- 4 players with 3 pure strategies each: 297 roots found in 4m3s220ms.
- 4 players with 4 pure strategies each: 13833 roots found in 7h2m20s780ms.
- 5 players with 2 pure strategies each: 44 roots found in 7s200ms.
- 6 players with 2 pure strategies each: 265 roots found in 7m10s790ms.

The running time seems to go up somewhat superlinearly with the Bernstein number (which may be considered part of the inherent complexity of the problem). Furthermore, this method is trivially parallelizable, requiring no communication between processors following different paths. For all but the smallest problems, it is the path-following which takes up most of the running time. For the smallest problems, the time used to compute the start system is significant. The start system, which depends only on the Newton polytope and thus can be the same for all games of a given format, could be precomputed.

6. OTHER DIRECTIONS

An interesting direction for further work is the computation of Nash equilibria under uncertainty. Specifically, the payoff functions may not be known exactly, but only approximately. A natural formulation is that each payoff value is known to lie in some interval. This leads to the question of how the set of Nash equilibria varies as the set of payoff values (now considered as parameters) varies.

Purely symbolically, such variations could be studied either through parametric Gröbner bases, as computed for example by Montes (2002) or Faugère (2002), or through resultants, as computed for example by Emiris and Canny (1996). For parametric Gröbner bases, the basic idea is to carry out the Gröbner basis computation, treating the payoff values as parameters and assuming that no cancellations ever occur. In this way one arrives at the generic solution. (A cancellation occurs whenever two algebraic expressions involving the parameters are equal. Thus, if desired, one can keep track of the algebraic equations assumed not to hold along the way, and thus determine in the end the algebraic subvariety of the payoff space which is *not* generic, as a by-product of this computation.) One might define resultants as the end result of such computations, but in fact resultants can be expressed much more compactly using determinantal formulas. In either case, if such formulas were precomputed for games of various formats, the Nash equilibria for any specific set of payoffs could be computed by evaluation of the formula in polynomial time (provided the payoffs were indeed generic; if not, division by zero would occur). Such formulas have been computed by Emiris in his thesis (1994) for very small cases, but unfortunately for larger cases. these computations are still intractable for the present.

A recent approach to finding real roots of polynomial systems is through semidefinite programming. Semialgebraic constraints can include nonnegativity constraints (such as arise in our problem) as well

as equations. These nonnegativity constraints are relaxed to the (sufficient) condition that the polynomials in question be the sums of squares of other polynomials (which of course are of lower degree). This condition can be expressed as the positive semidefiniteness of a matrix, namely the Gram matrix, which represents the quadratic form in the smaller polynomials in the monomial basis. Unfortunately we were not able to test the primary exemplars of this approach, SOStools and Gloptipoly, which require particular versions of Matlab. However, it is clear that this approach lends itself easily to the formulation of such “robust” computations. One uses parametric values for the payoffs and adds the constraints on the payoffs (for example, that they lie in a certain interval) to the problem.

As was discussed earlier, PHC succeeds at finding all the “quasi-equilibrium” points, and the result of McKelvey and McLennan (1997) shows that these may all be actual Nash equilibria. Thus, there is no way to avoid worst-case complexity given by the Bernstein number. However, in practice it will often be the case that many of the “quasi-equilibrium” points do not lie in the product of simplices, or are not even real. Practically, time would be saved by heuristic methods for examining the starting system and determining that a significant subset of the paths will not converge to a solution in the product of simplices and so do not need to be followed. Such heuristic methods have yet to be defined.

Game theorists generally would prefer that there be one distinguished equilibrium point for any particular game. Not only does this allow the game theorists to predict what will happen during the game, it allows the players themselves to predict what the other players will do. After all, if different players have different equilibrium points in mind when choosing their strategies, then the resulting behavior may not even be at equilibrium. For this reason various refinements of the Nash equilibrium concept have been proposed; these are summarized in *Stability and Perfection of Nash Equilibria*, by Van Damme (1991). In the past algebraic techniques have been used to find all Nash equilibria, and other techniques have been used to try to find a single Nash equilibrium (hopefully the “best” one in one of these senses). It would be interesting to see if the methods we have used can be modified to compute these more refined equilibria.

7. CONCLUSION

The main computer package for studying game theory today is Gambit. Although there are many ways to characterize Nash equilibria, the

one which lends itself most easily to the computation of *all* Nash equilibria of a game with more than two players is as solutions to systems of polynomial equations. However, the algorithm currently implemented in Gambit could be outperformed by the existing polyhedral homotopy continuation software PHC. So hopefully PHC or some similar package will soon be incorporated into Gambit. Furthermore, there are many other promising directions to pursue in applying algebra to game theory.

8. APPENDIX I: CODE LISTINGS

This appendix lists some of the utility code that was used to facilitate working with these games. The main file is `rorandgame.ml`. This provides a data structure to hold games (which are tables of numbers). It can generate random games of specified format and output them in files readable by Gambit. Moreover it can generate the polynomial system corresponding to a game and output these polynomials to a file.

```
(* Import the Numerix library *)
open Numerix
(* Import the needed modules from Dedicas *)
open RsdObject
open RsdRat
open RsdField
open RsdTermOrder
open RsdSubst
open RsdFloat

(* Declare the abstract datatype of normal form games *)
module type NORMAL_FORM_GAME =
sig
  (* The abstract data type itself *)
  type t
  (* The type of the entries (numbers) *)
  type payoff_t
  (* get_num_players game returns an integer, the number of players *)
  val get_num_players: t -> int
  (* create_from_num_players n returns a game with n players, 2
  strategies each; if n < 0, raises exception Invalid_argument *)
  val create_from_num_players: int -> t
  (* get_num_strategies game i returns the number of strategies
  player i has in game; raises Invalid_argument if i is not a valid
```

```

index *)
val get_num_strategies: t -> int -> int
(* set_num_strategies game i di sets the number of player i's
strategies in game to di; raises Invalid_argument if i is not a
valid index or di < 0. If di differs from its previous value, the
previous payoffs are lost. *)
val set_num_strategies: t -> int -> int -> unit
(* get_nums_strategies game returns an array [|dl;...;dn|] where
di is the number of player i's strategies in game *)
val get_nums_strategies: t -> int array
(* set_nums_strategies game [| dl,...,dn |] sets the number of
player i's strategies in game to di, and raises Invalid_argument
if di < 0 for any i or if the number of players is less than n *)
val set_nums_strategies: t -> int array -> unit
(* create_from_format n [| dl,...,dm |] returns a game with n
players, with dl,...,dn strategies respectively; raises
Invalid_argument if n < 0 or m < n *)
val create_from_format: int -> int array -> t
(* get_payoff game i [| j1,...,jn |] returns the payoff for player
i in game when player 1 chooses strategy j1,..., player n chooses
strategy jn. Raises Invalid_argument if i is not a player, if n
is less than the number of players, or if player k has less than
jk + 1 strategies (the strategies are indexed from 0). *)
val get_payoff: t -> int -> int array -> payoff_t
(* This function is just like string_of_payoff, but returns the
payoff as a string rather than a number. *)
val string_of_payoff: t -> int -> int array -> string
(* set_payoff game i [| j1,...,jn |] c sets the payoff for player
i in game to c when player 1 chooses strategy j1,..., player n
chooses strategy jn. Raises Invalid_argument if i is not a
player, if n is less than the number of players, or if player k
has less than jk + 1 strategies (the strategies are indexed from
0). *)
val set_payoff: t -> int -> int array -> payoff_t -> unit
end

(* Given a module with signature FIELD (that is, a mathematical field),
this functor returns a module with signature NORMAL_FORM_GAME, of games
with entries from that field. *)
module NormalFormGame( PayoffSpace : FIELD ) : NORMAL_FORM_GAME
  with type payoff_t = PayoffSpace.t =
  struct

```

```

type payoff_t = PayoffSpace.t
type t = int * int array * ( payoff_t array ) array
let get_num_players ( n, _, _ ) = n
let create_from_num_players n =
  ( n, Array.make n 1, Array.make n [| PayoffSpace.zero |] )
let get_num_strategies ( _, a, _ ) i = a.( i )
let get_nums_strategies ( _, a, _ ) = a
let init_payoffs n a p =
  let rec loop cum i =
    if i = n then cum else
      loop ( cum * a.( i ) ) ( i + 1 )
  in
  let total_d = loop 1 0 in
  let rec make_payoff_tensor j =
    if j = n then () else
      let _ = p.( j ) <- Array.make total_d PayoffSpace.zero in
      make_payoff_tensor ( j + 1 )
  in make_payoff_tensor 0
let set_num_strategies ( n, a, p ) i d =
  if d = a.( i ) then () else
    let _ = a.( i ) <- d in
    init_payoffs n a p
let set_nums_strategies ( n, a, p ) new_a =
  let rec loop i =
    if i = n then () else
      let _ = a.( i ) <- new_a.( i ) in loop ( i + 1 )
  in
  let _ = loop 0 in
  init_payoffs n a p
let create_from_format n a =
  let p = Array.make n [| PayoffSpace.zero |] in
  let _ = init_payoffs n a p in
  ( n, a, p )
let index ( n, a, p ) indices =
  let rec loop player i =
    if player = n - 1 then i else
      loop ( player + 1 ) ( i * a.( player ) + indices.( player + 1 )
  in loop 0 ( indices.( 0 ) )
let get_payoff ( n, a, p ) i indices =
  p.( i - 1 ).( index ( n, a, p ) indices )
let string_of_payoff g i indices =
  "(" ^ PayoffSpace.string_of ( get_payoff g i indices ) ^ ")"

```

```

    let set_payoff ( n, a, p ) i indices payoff =
      p.( i - 1 ).( index ( n, a, p ) indices ) <- payoff
    end

module type NORMAL_FORM_GAME_UI =
  sig
    module Game: NORMAL_FORM_GAME
      (* Through stdin/stdout, read_game_format () reads the number of
         players and the number of strategies for each player, and returns a
         game *)
      val read_game_format: unit -> Game.t
      (* randomizeallpayoffs game randomizes all the entries in game *)
      val randomizeallpayoffs: Game.t -> unit
      (* makerandomgame n d creates a game with n players, each with d
         strategies; it raises Invalid_argument if n < 0 or d < 0 *)
      val makerandomgame: int -> int -> Game.t
      (* writePHC game filename computes the
         system of totally mixed equations corresponding to game and
         outputs it to filename in the format expected by PHC. *)
      val writePHC: Game.t -> string -> unit
      (* writeGambit game filename writes a game to filename.nfg in the
         format expected by Gambit. *)
      val writeGambit: Game.t -> string -> unit
      (* writeAll game filename writes a game to files filename[.ext]
         for all the supported extensions, currently only null (for PHC)
         and .nfg (for Gambit. *)
      val writeAll: Game.t -> string -> unit
      (* writerandomgame prefix n d generates a random game with n
         players with d strategies each, and writes it to files
         <prefix><n>_<d>each.* for all the supported file formats *)
      val writerandomgame: string -> int -> int -> unit
    end

  (* This functor takes a module of signature FIELD, and a function
     random which takes an argument of type unit (just a placeholder) and
     returns an element of the field, and returns a module of signature
     NORMAL_FORM_GAME_UI *)
  module NormalFormGameUI( Arg:
    sig module PayoffSpace: FIELD
      val random: unit -> PayoffSpace.t
    end
  ) =

```

```

struct
  module PolySubst = SparsePolynomialWithSubstitutions( Arg.PayoffSpace
    ( LexicographicTermOrder )
  module PolySubstRep = RepPrintableObject( struct
    module Obj = PolySubst module Lex = Mainlexer end )
  module Game = NormalFormGame( Arg.PayoffSpace )
  let makepoly str =
    Grammar.Entry.parse PolySubstRep.main_entry ( Stream.of_string str )
  let read_game_format () =
    let _ = Printf.printf "How many players are in the game? " in
    let n = read_int () in
    let read_nums_strategies () =
      let rec loop i ds =
        if i = n then Array.of_list ( List.rev ds ) else begin
          let _ = Printf.printf "Please enter the number " in
          let _ = Printf.printf "of player %d's pure strategies:\n" (i + 1) in
          let d = read_int () in
          loop ( i + 1 ) ( d :: ds )
        end
      in loop 0 []
    in
    Game.create_from_format n ( read_nums_strategies () )
  let randomizeallpayoffs g =
    let n = Game.get_num_players g in
    let a = Game.get_nums_strategies g in
    let indices = Array.make n 0 in
    let rec loop i =
      if i > n then () else begin
        let _ = Game.set_payoff g i indices ( Arg.random () ) in
        let rec next_index j =
          if j = -1 then false else
            if indices.( j ) < a.( j ) - 1 then
              let _ = indices.( j ) <- indices.( j ) + 1 in true
            else
              let _ = indices.( j ) <- 0 in next_index ( j - 1 )
          in
          if next_index ( n - 1 ) then loop i else loop ( i + 1 )
        end
      in loop 1
    let makerandomgame num_players num_pure_strategies =
      let nums_strategies = Array.make num_players num_pure_strategies in
      let g = Game.create_from_format num_players nums_strategies in

```

```

let _ = randomizeallpayoffs g in
g
let writePHC g filename =
let gettotallymixedequations g =
let n = Game.get_num_players g in
let a = Game.get_nums_strategies g in
let vars =
let make_vars i =
let make_var j =
let varstr =
Stream.of_string ( Printf.sprintf "p%ds%d" ( i + 1 ) j )
in
Grammar.Entry.parse PolySubstRep.main_entry varstr
in
let pis = Array.init a.( i ) make_var in
let rec sumvars sum j =
if j = a.( i ) then sum else
sumvars ( PolySubst.Rule.Poly.add sum pis.( j ) ) ( j + 1
in
let _ =
pis.( 0 ) <-
( PolySubst.Rule.Poly.subtract
PolySubst.Rule.Poly.one
( sumvars PolySubst.Rule.Poly.zero 1 ) )
in pis
in Array.init n make_vars
in
let indices = Array.make n 0 in
let indices0 = Array.make n 0 in
let rec playerloop i eqns =
if i = n then eqns else
let rec purestrategyloop j eqnssofar =
if j = a.( i ) then eqnssofar else begin
let rec init_indices k =
if k = n then () else
let _ =
if k = i then indices.( k ) <- j
else indices.( k ) <- 0 in
let _ = indices0.( k ) <- 0 in
init_indices ( k + 1 )
in
let _ = init_indices 0 in

```

```

let rec termloop poly =
  let payoff1 = Game.get_payoff g ( i + 1 ) indices in
  let payoff0 = Game.get_payoff g ( i + 1 ) indices0 in
  let coeff = Arg.PayoffSpace.subtract payoff1 payoff0 in
  let rec make_term k m =
    if k = n then m else begin
      if k = i then make_term ( k + 1 ) m else
        let var = vars.( k ).( indices.( k ) ) in
        let newm = PolySubst.Rule.Poly.mult m var in
        make_term ( k + 1 ) newm
    end
  end
  in
  let term = make_term 0 ( PolySubst.Rule.Poly.of_const co ) in
  let newpoly = PolySubst.Rule.Poly.add poly term in
  let rec next_index k =
    if k = -1 then false else begin
      if k = i then next_index ( k - 1 ) else begin
        if indices.( k ) = a.( k ) - 1 then
          let _ = indices.( k ) <- 0 in
          let _ = indices0.( k ) <- 0 in
          next_index ( k - 1 )
        else
          let _ = indices.( k ) <- indices.( k ) + 1 in
          let _ = indices0.( k ) <- indices0.( k ) + 1 in
          true
        end
      end
    end
  end
  in
  if next_index ( n - 1 ) then termloop newpoly else newpoly
in
let neweqn = termloop ( PolySubst.Rule.Poly.zero ) in
purestrategyloop ( j + 1 ) ( neweqn :: eqnssofar )
end
in
let neweqns = purestrategyloop 1 eqns in
playerloop ( i + 1 ) neweqns
in List.rev ( playerloop 0 [] )
in
let eqns = gettotallymixedequations g dehomogenize in
let oc = open_out filename in
let _ = Printf.fprintf oc "%d\n" ( List.length eqns ) in
let pr_eq eqn =

```

```

Printf.fprintf oc " %s;\n" ( PolySubst.Rule.Poly.string_of eqn )
in
let _ = List.iter pr_eq eqns in
close_out oc
let writeGambit g filename =
  let num_players = Game.get_num_players g in
  let oc = open_out ( filename ^ ".nfg" ) in
  let _ = Printf.fprintf oc "NFG l R \"Untitled Normal Form Game\" { "
  let rec loop_names n =
    if n > num_players then () else
      let _ = Printf.fprintf oc "\"Player%d\" " n in loop_names ( n +
in
let _ = loop_names 1 in
let _ = Printf.fprintf oc "}\n\n{ " in
let rec loop_strategynames n =
  if n > num_players then () else
    let _ = Printf.fprintf oc "{ " in
    let snum = Game.get_num_strategies g n in
    let rec loop_s s =
      if s > snum then () else
        let _ = Printf.fprintf oc "\"%d\" " s in loop_s ( s + 1 )
    in
    let _ = loop_s 1 in
    let _ = Printf.fprintf oc "}\n\n"
    in loop_strategynames ( n + 1 )
in
let _ = loop_strategynames 1 in
let _ = Printf.fprintf oc "}\n\n\""\n\n{" in
let outcome = Array.make num_players 1 in
let rec incr_outcome i =
  if i < 1 then false
  else if outcome.( i ) < Game.get_num_strategies g i then
    let _ = outcome.( i ) <- outcome.( i ) + 1 in true
  else incr_outcome ( i - 1 )
in
let rec loop_playeractions i =
  if i > num_players then () else
    let _ = Printf.fprintf oc "%d" outcome.( i ) in
    loop_playeractions ( i + 1 )
in
let rec loop_playerpayoffs i =
  if i > num_players then () else begin

```

```

        let _ = Printf.fprintf oc "%s" ( Game.string_of_payoff g i outcome )
        let _ = if i < num_players then Printf.fprintf oc "," else () in
        let _ = Printf.fprintf oc " " in
        loop_playerpayoffs ( i + 1 )
    end
in
let rec print_payoffs c =
    let _ = Printf.fprintf oc "{ \" in
    let _ = loop_playeractions 1 in
    let _ = Printf.fprintf oc "\" " in
    let _ = loop_playerpayoffs 1 in
    let _ = Printf.fprintf oc "}\n" in
    if incr_outcome num_players then print_payoffs ( c + 1 ) else c
in
let num_contingencies = print_payoffs 1 in
let _ = Printf.fprintf oc "}\n" in
let rec loop_results c =
    if c > num_contingencies then () else
        let _ = Printf.fprintf oc "0 " in loop_results ( c + 1 )
in
close_out oc
let writeAll g filename =
    let _ = writePHC g filename in
    let _ = writeGambit g filename in
    ()
let writerandomgame prefix num_players num_strategies =
    let g = makerandomgame num_players num_strategies in
    let filename =
        ( Printf.sprintf "%s%d_%deach" prefix num_players num_strategies )
    in
    writeAll g filename
end

```

```

module FloatNormalFormGameUI
= NormalFormGameUI(
    struct
        module PayoffSpace = Float
        let random () = Random.float 256.0
    end )

```

```

module RationalNormalFormGameUI
= NormalFormGameUI(

```

```

struct
  module PayoffSpace = MultiPrecRat( Numerix.Slong )
  let random () = ( Slong.of_int ( Random.int 256 ), Slong.of_int 1 )
end )

```

The game utility code uses a polynomial arithmetic package that we wrote, called Dedicas. Both the utility code and the Dedicas are written in Ocaml. Ocaml is a general-purpose language with strong static typing and support for functional, object-oriented and imperative programming. It includes a convenient interactive loop. In our experience, the typechecker catches most of the bugs we typically make. Once a program compiles successfully, there are very few bugs. The benefits of typechecking extend to the module system, which provides data hiding and encapsulation. (Although Ocaml also supports object-oriented programming, for our purposes modules subdivide our program cleanly without the drawbacks of dynamic dispatch.) Since the computations of interest are to be done by other programs, the performance of our utility code is not an issue. We use Ocaml for its convenience in allowing us to quickly write correct programs. We wrote Dedicas for use in this and other projects, since polynomial arithmetic is not a part of Ocaml.

Dedicas makes use of the `camp4` parsing package and the Numerix package for arbitrary-precision integer arithmetic. Numerix provides a single interface to one of three multiprecision arithmetic libraries: Gnu's GMP, Ocaml's `Big_int`, and Numerix's own implementations. We used Numerix's `Slong`. Polynomials are collections of monomials stored in an ordered dictionary. We used an abstract datatype of ordered dictionaries, which we have implemented both as a balanced binary tree and as a hash table (which we currently use).

Here we present the file `rorandgame.ml`. It is also available on the web, at <http://www.math.berkeley.edu/~datta/rorandgame.ml>. The type `unit` contains only one element, called `()`. It is returned by functions which are actually procedures (i.e., which are called only for their side effects).

```

(* Import the Numerix library *)
open Numerix
(* Import the needed modules from Dedicas *)
open RsdObject
open RsdRat
open RsdField
open RsdTermOrder
open RsdSubst

```

```
open RsdFloat
```

```
(* Declare the abstract datatype of normal form games *)
```

```
module type NORMAL_FORM_GAME =
```

```
sig
```

```
  (* The abstract data type itself *)
```

```
  type t
```

```
  (* The type of the entries (numbers) *)
```

```
  type payoff_t
```

```
  (* get_num_players game returns an integer, the number of players *)
```

```
  val get_num_players: t -> int
```

```
  (* create_from_num_players n returns a game with n players, 2 strategies each; if n < 0, raises exception Invalid_argument *)
```

```
  val create_from_num_players: int -> t
```

```
  (* get_num_strategies game i returns the number of strategies player i has in game; raises Invalid_argument if i is not a valid index *)
```

```
  val get_num_strategies: t -> int -> int
```

```
  (* set_num_strategies game i di sets the number of player i's strategies in game to di; raises Invalid_argument if i is not a valid index or di < 0. If di differs from its previous value, the previous payoffs are lost. *)
```

```
  val set_num_strategies: t -> int -> int -> unit
```

```
  (* get_nums_strategies game returns an array [|d1;...;dn|] where di is the number of player i's strategies in game *)
```

```
  val get_nums_strategies: t -> int array
```

```
  (* set_nums_strategies game [| d1,...,dn |] sets the number of player i's strategies in game to di, and raises Invalid_argument if di < 0 for any i or if the number of players is less than n *)
```

```
  val set_nums_strategies: t -> int array -> unit
```

```
  (* create_from_format n [| d1,...,dm |] returns a game with n players, with d1,...,dn strategies respectively; raises Invalid_argument if n < 0 or m < n *)
```

```
  val create_from_format: int -> int array -> t
```

```
  (* get_payoff game i [| j1,...,jn |] returns the payoff for player i in game when player 1 chooses strategy j1,..., player n chooses strategy jn. Raises Invalid_argument if i is not a player, if n is less than the number of players, or if player k has less than jk + 1 strategies (the strategies are indexed from 0). *)
```

```
  val get_payoff: t -> int -> int array -> payoff_t
```

```
  (* This function is just like string_of_payoff, but returns the
```

```

payoff as a string rather than a number. *)
val string_of_payoff: t -> int -> int array -> string
(* set_payoff game i [| j1,...,jn |] c sets the payoff for player
i in game to c when player 1 chooses strategy j1,..., player n
chooses strategy jn. Raises Invalid_argument if i is not a
player, if n is less than the number of players, or if player k
has less than jk + 1 strategies (the strategies are indexed from
0). *)
val set_payoff: t -> int -> int array -> payoff_t -> unit
end

```

(* Given a module with signature FIELD (that is, a mathematical field), this functor returns a module with signature NORMAL_FORM_GAME, of games with entries from that field. *)

```

module NormalFormGame( PayoffSpace : FIELD ) : NORMAL_FORM_GAME
with type payoff_t = PayoffSpace.t =
struct
  type payoff_t = PayoffSpace.t
  type t = int * int array * ( payoff_t array ) array
  let get_num_players ( n, _, _ ) = n
  let create_from_num_players n =
    ( n, Array.make n 1, Array.make n [| PayoffSpace.zero |] )
  let get_num_strategies ( _, a, _ ) i = a.( i )
  let get_nums_strategies ( _, a, _ ) = a
  let init_payoffs n a p =
    let rec loop cum i =
      if i = n then cum else
        loop ( cum * a.( i ) ) ( i + 1 )
    in
    let total_d = loop 1 0 in
    let rec make_payoff_tensor j =
      if j = n then () else
        let _ = p.( j ) <- Array.make total_d PayoffSpace.zero in
        make_payoff_tensor ( j + 1 )
    in make_payoff_tensor 0
  let set_num_strategies ( n, a, p ) i d =
    if d = a.( i ) then () else
      let _ = a.( i ) <- d in
      init_payoffs n a p
  let set_nums_strategies ( n, a, p ) new_a =
    let rec loop i =
      if i = n then () else

```

```

        let _ = a.( i ) <- new_a.( i ) in loop ( i + 1 )
    in
    let _ = loop 0 in
    init_payoffs n a p
let create_from_format n a =
    let p = Array.make n [| PayoffSpace.zero |] in
    let _ = init_payoffs n a p in
    ( n, a, p )
let index ( n, a, p ) indices =
    let rec loop player i =
        if player = n - 1 then i else
            loop ( player + 1 ) ( i * a.( player ) + indices.( player + 1 )
    in loop 0 ( indices.( 0 ) )
let get_payoff ( n, a, p ) i indices =
    p.( i - 1 ).( index ( n, a, p ) indices )
let string_of_payoff g i indices =
    "(" ^ PayoffSpace.string_of ( get_payoff g i indices ) ^ ")"
let set_payoff ( n, a, p ) i indices payoff =
    p.( i - 1 ).( index ( n, a, p ) indices ) <- payoff
end

```

```

module type NORMAL_FORM_GAME_UI =
sig
    module Game: NORMAL_FORM_GAME
    (* Through stdin/stdout, read_game_format () reads the number of
    players and the number of strategies for each player, and returns a
    game *)
    val read_game_format: unit -> Game.t
    (* randomizeallpayoffs game randomizes all the entries in game *)
    val randomizeallpayoffs: Game.t -> unit
    (* makerandomgame n d creates a game with n players, each with d
    strategies; it raises Invalid_argument if n < 0 or d < 0 *)
    val makerandomgame: int -> int -> Game.t
    (* writePHC game filename computes the
    system of totally mixed equations corresponding to game and
    outputs it to filename in the format expected by PHC. *)
    val writePHC: Game.t -> string -> unit
    (* writeGambit game filename writes a game to filename.nfg in the
    format expected by Gambit. *)
    val writeGambit: Game.t -> string -> unit
    (* writeAll game filename writes a game to files filename[.ext]
    for all the supported extensions, currently only null (for PHC)

```

```

and .nfg (for Gambit. *)
val writeAll: Game.t -> string -> unit
(* writerandomgame prefix n d generates a random game with n
players with d strategies each, and writes it to files
<prefix><n>_<d>each.* for all the supported file formats *)
val writerandomgame: string -> int -> int -> unit
end

(* This functor takes a module of signature FIELD, and a function
random which takes an argument of type unit (just a placeholder) and
returns an element of the field, and returns a module of signature
NORMAL_FORM_GAME_UI *)
module NormalFormGameUI( Arg:
  sig module PayoffSpace: FIELD
    val random: unit -> PayoffSpace.t
  end
) =
struct
  module PolySubst = SparsePolynomialWithSubstitutions( Arg.PayoffSpace
    ( LexicographicTermOrder )
  module PolySubstRep = RepPrintableObject( struct
    module Obj = PolySubst module Lex = Mainlexer end )
  module Game = NormalFormGame( Arg.PayoffSpace )
  let makepoly str =
    Grammar.Entry.parse PolySubstRep.main_entry ( Stream.of_string str )
  let read_game_format () =
    let _ = Printf.printf "How many players are in the game? " in
    let n = read_int () in
    let read_nums_strategies () =
      let rec loop i ds =
        if i = n then Array.of_list ( List.rev ds ) else begin
          let _ = Printf.printf "Please enter the number " in
          let _ = Printf.printf "of player %d's pure strategies:\n" (i +
            let d = read_int () in
            loop ( i + 1 ) ( d :: ds )
        end
      in loop 0 []
    in
    Game.create_from_format n ( read_nums_strategies () )
  let randomizeallpayoffs g =
    let n = Game.get_num_players g in
    let a = Game.get_nums_strategies g in

```

```

let indices = Array.make n 0 in
let rec loop i =
  if i > n then () else begin
    let _ = Game.set_payoff g i indices ( Arg.random () ) in
    let rec next_index j =
      if j = -1 then false else
        if indices.( j ) < a.( j ) - 1 then
          let _ = indices.( j ) <- indices.( j ) + 1 in true
        else
          let _ = indices.( j ) <- 0 in next_index ( j - 1 )
    in
    if next_index ( n - 1 ) then loop i else loop ( i + 1 )
  end
in loop 1
let makerandomgame num_players num_pure_strategies =
  let nums_strategies = Array.make num_players num_pure_strategies in
  let g = Game.create_from_format num_players nums_strategies in
  let _ = randomizeallpayoffs g in
  g
let writePHC g filename =
  let gettotallymixedequations g =
    let n = Game.get_num_players g in
    let a = Game.get_nums_strategies g in
    let vars =
      let make_vars i =
        let make_var j =
          let varstr =
            Stream.of_string ( Printf.sprintf "p%d%s%d" ( i + 1 ) j )
          in
          Grammar.Entry.parse PolySubstRep.main_entry varstr
        in
        let pis = Array.init a.( i ) make_var in
        let rec sumvars sum j =
          if j = a.( i ) then sum else
            sumvars ( PolySubst.Rule.Poly.add sum pis.( j ) ) ( j + 1 )
        in
        let _ =
          pis.( 0 ) <-
            ( PolySubst.Rule.Poly.subtract
              PolySubst.Rule.Poly.one
              ( sumvars PolySubst.Rule.Poly.zero 1 ) )
        in pis
    in

```

```

    in Array.init n make_vars
  in
  let indices = Array.make n 0 in
  let indices0 = Array.make n 0 in
  let rec playerloop i eqns =
    if i = n then eqns else
      let rec purestrategyloop j eqnssofar =
        if j = a.( i ) then eqnssofar else begin
          let rec init_indices k =
            if k = n then () else
              let _ =
                if k = i then indices.( k ) <- j
                  else indices.( k ) <- 0 in
              let _ = indices0.( k ) <- 0 in
              init_indices ( k + 1 )
            in
          let _ = init_indices 0 in
          let rec termloop poly =
            let payoff1 = Game.get_payoff g ( i + 1 ) indices in
            let payoff0 = Game.get_payoff g ( i + 1 ) indices0 in
            let coeff = Arg.PayoffSpace.subtract payoff1 payoff0 in
            let rec make_term k m =
              if k = n then m else begin
                if k = i then make_term ( k + 1 ) m else
                  let var = vars.( k ).( indices.( k ) ) in
                  let newm = PolySubst.Rule.Poly.mult m var in
                  make_term ( k + 1 ) newm
                end
              in
            let term = make_term 0 ( PolySubst.Rule.Poly.of_const co
            let newpoly = PolySubst.Rule.Poly.add poly term in
            let rec next_index k =
              if k = -1 then false else begin
                if k = i then next_index ( k - 1 ) else begin
                  if indices.( k ) = a.( k ) - 1 then
                    let _ = indices.( k ) <- 0 in
                    let _ = indices0.( k ) <- 0 in
                    next_index ( k - 1 )
                  else
                    let _ = indices.( k ) <- indices.( k ) + 1 in
                    let _ = indices0.( k ) <- indices0.( k ) + 1 in
                    true

```

```

        end
      end
    in
      if next_index ( n - 1 ) then termloop newpoly else newpo
    in
      let neweqn = termloop ( PolySubst.Rule.Poly.zero ) in
      purestrategyloop ( j + 1 ) ( neweqn :: eqnssofar )
    end
  end
in
  let neweqns = purestrategyloop 1 eqns in
  playerloop ( i + 1 ) neweqns
in List.rev ( playerloop 0 [] )
in
let eqns = gettotallymixedequations g dehomogenize in
let oc = open_out filename in
let _ = Printf.fprintf oc "%d\n" ( List.length eqns ) in
let pr_eq eqn =
  Printf.fprintf oc " %s;\n" ( PolySubst.Rule.Poly.string_of eqn )
in
let _ = List.iter pr_eq eqns in
close_out oc
let writeGambit g filename =
  let num_players = Game.get_num_players g in
  let oc = open_out ( filename ^ ".nfg" ) in
  let _ = Printf.fprintf oc "NFG 1 R \"Untitled Normal Form Game\" { "
  let rec loop_names n =
    if n > num_players then () else
      let _ = Printf.fprintf oc "\"Player%d\" " n in loop_names ( n +
  in
  let _ = loop_names 1 in
  let _ = Printf.fprintf oc "}\n\n{ " in
  let rec loop_strategynames n =
    if n > num_players then () else
      let _ = Printf.fprintf oc "{ " in
      let snum = Game.get_num_strategies g n in
      let rec loop_s s =
        if s > snum then () else
          let _ = Printf.fprintf oc "\"%d\" " s in loop_s ( s + 1 )
      in
      let _ = loop_s 1 in
      let _ = Printf.fprintf oc "}\n"
      in loop_strategynames ( n + 1 )

```

```

in
let _ = loop_strategynames l in
let _ = Printf.fprintf oc "}\n\""\n\n{" in
let outcome = Array.make num_players l in
let rec incr_outcome i =
  if i < l then false
  else if outcome.( i ) < Game.get_num_strategies g i then
    let _ = outcome.( i ) <- outcome.( i ) + 1 in true
  else incr_outcome ( i - 1 )
in
let rec loop_playeractions i =
  if i > num_players then () else
    let _ = Printf.fprintf oc "%d" outcome.( i ) in
    loop_playeractions ( i + 1 )
in
let rec loop_playerpayoffs i =
  if i > num_players then () else begin
    let _ = Printf.fprintf oc "%s" ( Game.string_of_payoff g i outco
    let _ = if i < num_players then Printf.fprintf oc "," else () in
    let _ = Printf.fprintf oc " " in
    loop_playerpayoffs ( i + 1 )
  end
in
let rec print_payoffs c =
  let _ = Printf.fprintf oc "{ \\" in
  let _ = loop_playeractions l in
  let _ = Printf.fprintf oc "\\" in
  let _ = loop_playerpayoffs l in
  let _ = Printf.fprintf oc "}\n" in
  if incr_outcome num_players then print_payoffs ( c + 1 ) else c
in
let num_contingencies = print_payoffs l in
let _ = Printf.fprintf oc "}\n" in
let rec loop_results c =
  if c > num_contingencies then () else
    let _ = Printf.fprintf oc "0 " in loop_results ( c + 1 )
in
close_out oc
let writeAll g filename =
  let _ = writePHC g filename in
  let _ = writeGambit g filename in
  ()

```

```

let writerandomgame prefix num_players num_strategies =
  let g = makerandomgame num_players num_strategies in
  let filename =
    ( Printf.sprintf "%s%d_%deach" prefix num_players num_strategies )
  in
  writeAll g filename
end

module FloatNormalFormGameUI
= NormalFormGameUI(
  struct
    module PayoffSpace = Float
    let random () = Random.float 256.0
  end )

module RationalNormalFormGameUI
= NormalFormGameUI(
  struct
    module PayoffSpace = MultiPrecRat( Numerix.Slong )
    let random () = ( Slong.of_int ( Random.int 256 ), Slong.of_int 1 )
  end )

```

Now to use this code, we start Ocaml using the command

```
ocamlnumx
```

to enter the interactive loop in which Numerix is preloaded. Then we type

```
#load "rsdDedicas.cma";;
```

to load Dedicas, and

```
#use "rorandgame.ml"
```

to input (and compile) our utility code. Finally, we type

```
RationalNormalFormGameUI.writerandomgame 3 2;;
```

to write Gambit and PHC input files for a random game.

9. APPENDIX II: INPUT FILES

Since game specifications grow very quickly with increasing numbers of players and strategies, we give here only one example of each input file, for a game with 3 players, each with 2 pure strategies. Here is the input file for Gambit (one, incidentally, which caused Gambit to crash):

```

NFG 1 R "Untitled Normal Form Game" { "Player1" "Player2" "Player3" }
{ { "1" "2" }

```

```

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

{
{ "111" 112, 12, 112 }
{ "112" 17, 24, 27 }
{ "121" 25, 46, 43 }
{ "122" 9, 117, 4 }
{ "211" 170, 232, 34 }
{ "212" 155, 78, 180 }
{ "221" 8, 173, 79 }
{ "222" 230, 218, 145 }
}
0 0 0 0 0 0 0 0

```

Here is the file we created to read into Macaulay2:

```

R = QQ[ p1s1, p2s1, p3s1, MonomialOrder=>Lex ]
g = ideal (
-158*p2s1*p3s1 + 75*p2s1 + -80*p3s1 + -58,
-140*p1s1*p3s1 + -59*p1s1 + 93*p3s1 + -34,
126*p1s1*p2s1 + -231*p1s1 + -46*p2s1 + 85 )

```

The variable `p2s1` indicates the probability that player 2 will pick strategy 1. (Unlike Gambit, we number our strategies from 0, so this is what Gambit sees as strategy 2.) We then issue the command

```
time gens gb g
```

in Macaulay2 to time the generation of a Gröbner basis.

The input file to be read into Singular is very similar:

```

ring r = 0, ( p1s1, p2s1, p3s1 ), lp;
ideal g = (
-158*p2s1*p3s1 + 75*p2s1 + -80*p3s1 + -58,
-140*p1s1*p3s1 + -59*p1s1 + 93*p3s1 + -34,
126*p1s1*p2s1 + -231*p1s1 + -46*p2s1 + 85 );

```

We had previously issued the command

```
LIB "solve.lib"
```

to load the standard library `solve.lib`. Now we issue the command

```
int t = timer; solve(groebner(g)); timer - t;
```

to time the generation of a Gröbner basis and numerical solution of the polynomial system.

The file, `intgame3_2each` to be read into PHC is even simpler:

3

```
-158*p2s1*p3s1 + 75*p2s1 - 80*p3s1 - 58;  
-140*p1s1*p3s1 - 59*p1s1 + 93*p3s1 - 34;  
126*p1s1*p2s1 - 231*p1s1 - 46*p2s1 + 85;
```

(The number 3 designates the number of equations.) Then we issue the command

```
phc -b intgame3_2each intgame3_2each.bphc
```

to find the roots of the system.