Fixed Points and Imitation Games

The `IG` Package

code by

Colin Ramsay

manual by

Andrew McLennan and Colin Ramsay

(Version 1.1: February 17, 2012)

This manual describes how to install and use version 1.1 of the `IG` (imitation game) software.

Contents

List of Figures

1 Introduction

This document describes version 1.1 of the `IG` package, which provides software for computing approximate fixed points based on the algorithm described in [9] for computing approximate fixed points of functions by building a series of imitation games and finding Nash equilibria of them. The code was written by Colin Ramsay. This manual was written by Andrew McLennan and Colin Ramsay. Work on the `IG` package was funded by Australian Research Council Discovery Project grant DP0773324.

Although the `IG` package is produced under the auspices of the Gambit project [8], it is independent of any extant Gambit code.

1.1 What Does the `IG` Package Do?

Brouwer's fixed point theorem asserts that if C is a nonempty compact convex subset of a finite dimensional space and $f : C \to C$ is a continuous function, then f has a *fixed point*, which is a point $x^* \in C$ such that $f(x^*) = x^*$. Walrasian equilibria economies and Nash equilibria of games can be understood as fixed points, so algorithms that compute approximate fixed points can be applied in computational work in economics. More generally, a fixed point can be thought of as a solution of a nonlinear system of equations with the same number of equations as variables. These arise in a wide variety of scientific contexts, so software of the sort presented by `IG` package has many application domains.

In [9] McLennan and Tourky described a computational strategy for finding approximate fixed points in which the function is sampled at a number of points, a two person game is constructed, the Lemke-Howson algorithm is used to find a Nash equilibrium of this game, and a weighted average of the points in the domain, with the weights given by the equilibrium mixed strategies, is a "candidate approximate fixed point." The `IG` package provides practical software that takes advantage of this theoretical advance, blending this approach with random search, as a way to get a favorable initial point of the algorithm, and with the Newton method, as a way to rapidly improve the accuracy once the algorithm is in a neighborhood of a fixed point.

The overall approach is based on floating-point computation, which (due to round-off error and limited accuracy) comes with few guarantees. The software design takes account of this by making multiple independent attempt, each of which is expected to fail gracefully when it does not succeed. The fixed point finder has many parameters that allow the user to "tune" it in response to experience in the user's application domain.

1.2 Required Resources

The `IG` package is written in the C programming language, in accord with the C99 standard, so the user must have a compiler supporting this standard. The code is built on top of the GNU Scientific Library, which must be available. Instructions for installing the GSL on Unix platforms are given in the next section. At a minimum the user must

be able to code a function of interest in C, and to write code that does what the user wants with the output. As with any software package, additional knowledge, ambition, and effort allows for more sophiticated utilization.

1.3 How the Rest of this Document is Structured

This manual is designed to support a process in which the reader first becomes acquainted with the simplest sorts of applications of `IG` package, and then learns more about the details of its design and the theoretical basis of the algorithms. The next section gives installation instructions, and walks through a simple example program. Some background concerning the theory of the computation of fixed points, other methods that might be used to find them, and the workings of the imitation game algorithm, are described in Section 3. Section 4 describes the code, emphasizing the aspects that are "visible" in the sense of being declared in the header files. Section 5 provides some information concerning floating-point issues and the GSL to users who might wish to modify the code to achieve more sophisticated algorithms or applications. Section 6 describes the example programs supplied with the `IG` package. These were used to test the package, serve as demonstrations of how it functions and how to write programs using it, and address the problems of defining your own user function and dealing with floating-point issues, in various ways. Finally, Appendix A lists all the command line options used by all the example programs, and provides a brief description of each one.

2 Getting Started

This section gives detailed and concrete instructions for how to install the `IG` package, and how to begin understanding what it does and how it works.

2.1 Package installation and use

Version 1.1 of the `IG` package is supplied as a single gzip'd tar archive `ig-1.1.tar.gz`. We will assume a UNIX environment; specifically, we have tested the setup under SunOS, MacOS, Ubuntu, and cygwin. After you have downloaded the file and placed it in a suitable directory, type the command line

```
> gunzip ig-1.1.tar.gz
```

This converts the file `ig-1.1.tar.gz` to the file `ig-1.1.tar`. Next, type the command line

```
> tar xvf ig-1.1.tar
```

Now, in addition to the file `ig-1.1.tar` (which is unchanged) there will be a directory `ig-1.1` which contains all the material that was unpacked. After

```
> cd ig-1.1
> ls
```

you will see the source code and a Makefile to build the example programs. This directory also contains a `doc` subdirectory which contains copies of this document (as `.dvi`, `.ps` and `.pdf` files) as well as its LaTeX source.

You will need the GNU Scientific Library. If it is not already installed on your system you can go to the internet and download either the latest version or `gsl-1.15.tar.gz` (if you suspect that the there is a problem that crept in with later versions). Put it into a suitable directory, then do

```
> gunzip gsl-1.15.tar.gz
> tar xvf gsl-1.15.tar
```

If you have a superuser password you can now do

```
> cd gsl-1.15
> ./configure
> make
> sudo make install
```

and after typing in your superuser password when prompted, `gsl` will be installed on the entire system. If you don't have a superuser password, but you are on good terms with the system administrator *and* she thinks it is a good idea, possibly you can cajole her into helping you install it systemwide. If one of these methods works you can now go back to the `ig-1.1` directory and do

```
> make
```

or

```
> make all
```

and (at least in our experience) the software should compile with *no* warnings.

If you don't have a superuser password and your sysadmin is unyielding, as a last resort you can try to use a personal installation of `gsl`. Here is how it worked for us. Suppose that you put `gsl-1.15.tar.gz` in a directory `/home/me/GSL` before `gunzip`'ing it and extracting its contents to the directory `gsl-1.15`. You can now do

```
> mkdir ins
> cd gsl-1.15
> ./configure -prefix="/home/me/GSL/ins"
> make
> make install
```

When you go back to the directory `ig-1.1` you will need to tell the `Makefile` for the `IG` package about the installation of `gsl`. The way to do this is to replace the lines

```
CFLAGS = -O3 -Wall -std=c99 -pedantic
LDFLAGS = -lm
```

with

```
CFLAGS = -O3 -Wall -std=c99 -pedantic -I/home/me/GSL/ins/include
LDFLAGS = -lm -L/home/me/GSL/ins/lib
```

After this, `make` may work without a hitch, but things are getting a little complex here, so who knows what problems might arise on your system. Good luck!

Note the use of the `-Wall`, `-std=c99` and `-pedantic` compiler switches. The `Makefile` is for GNU/Linux systems, and may need tweaking for other systems. Any "standard compliant" C compiler can be used instead of GNU's `gcc`. On some systems, the include and library paths for GSL may need to be included using the `-I...` and `-L...` compiler flags, as above, and the `LD_LIBRARY_PATH` environment variable set to enable the executable to be run.

2.2 An Example Program

The `IG` package is a collection of C functions that are used to find fixed points of functions specified by the user, and the user will need to write code that specifies the function, as well as what to do with the fixed points that are found. If you don't care to understand just how the fixed points are computed, you can ignore much of the rest of this document, but for practical purposes the knowledge and skills developed in this section are crucial.

We are going to *very slowly* walk through the example program `demo1`. Most likely, your own use of the `IG` package will be primarily a matter of modifying the source code

Figure 1: Example output of the `demo1` program

```
Found a fixed point, norm = 9.45938737062502e-07
  X[0] = 0.5  Y[0] = 0.5
  X[1] = 0.414213562373095  Y[1] = 0.414213562373095
  X[2] = 0.707106781186546  Y[2] = 0.707106781186549
  X[3] = 0.292893691782979  Y[3] = 0.292892745844242
```

of this (or another of the example programs) to suit your purposes. The explanation will be pitched to a newcomer to C, but this is at least a bit unrealistic. If you're like me, you don't learn a new programming language by taking a course or reading a book on it, but instead by trying to write a program in it, looking at the book primarily when searching for specific information. Everyone who takes this approach has to start somewhere, but as a starting point for C the the `IG` package environment is pretty advanced. If you really are new to C (or relatively inexperienced) you may think of this as a rough description of concepts you will need to study more carefully in order to really work with the `IG` package. Mainly we have assumed such a low level audience in order to be as inclusive as possible (we hope without insulting the vast majority, who already know a lot about C) and it also has the very real advantage of slowing things down, forcing us to examine each detail.

If you did `make` or `make all`, as instructed above, and everything went smoothly, you should have an executable called `demo1` in the directory `ig-1.1`. To run it do

```
> ./demo1
```

(Here `./` refers to the current directory. If `.` is in your PATH you can simply type `demo1`.) The output should look something like Figure 1. There is some randomization going on inside, as you'll see if you run `demo1` again, so the details will almost certainly be different. Here `X` and `Y` are four-dimensional vectors. (Newbies should note that in C the indexing of vectors, arrays, and matrices (which are arrays of arrays) always starts with 0.) As the notation suggests, `X` is the argument of a function, `Y` is the value of the function, and `norm` is the distance from `X` to `Y`.

Let's now look at the source code for `demo1`'s `main` (the master controller of any C program) which is shown in Figure 2. In C every variable has a type, which may be user defined, or a builtin like `int` and `char` (integer and character respectively). The first line says that `main` is a *function* (C's jargon for procedure or subroutine) whose return value (which should not be confused with the consequences of running it!) is an `int` (usually a code for the exit status) and whose inputs are an `int` called `argc` and a `*char[]` (roughly, a list of strings) called `argv`. Typically these refer to arguments that are passed to the program from the command line; we won't worry about them at this point.

Going inside the definition of `main`, we first see three lines that declare three variables. In C you always have to declare a variable, which amounts to saying what its type is, before you can do anything with it. Here the second line says that `igfps` is a variable whose type is `igFPSTAT`, and the third line say that `i` is a variable whose type is `int`. In C programming a standard style is to give short names like `i`, `d`, and `s` to variables that are used to do simple things within a local context, for example indexing

Figure 2: The `main` function of the `demo1` program

```
int main(int argc, char *argv[])
  {
  igList *igl;
  igFPSTAT igfps;
  int i;

  igigSetup();

  igDim = 4;
  igFunction = *f1Apply;
  igigDmin = 0.0;
  igigDmax = 1.0;

  igl = igNewList(igDim);
  igfps = igigFindFP(igl, 0U);

  if (igfps == igFail)
    { printf("Unable to find any (approx) fixed point\n"); }
  else
    {
    if (igfps == igFP)
      { printf("Found a fixed point, norm = %0.15g\n", igl->dn[igl->size-1]); }
    else
      { printf("Best approx'n found, norm = %0.15g\n", igl->dn[igl->size-1]); }

    for (i = 0; i < igDim; i++)
      { printf("  X[%d] = %0.15g  Y[%d] = %0.15g\n",
                      i, igl->dx[igl->size-1][i], i, igl->dy[igl->size-1][i]); }
    }

  free(igl);
  exit(EXIT_SUCCESS);
  }
```

iteration in loops and such. Another standard style for largish packages is to use a prefix for all types, variables, and functions that are defined within the package, at least if they might be referred to by code that uses the package. The `IG` package uses `ig` to tag the objects it defines, and `igList` and `igFPSTAT` are two such types.

What's going on with the `*`? Any variable that occurs in a C program has three aspects: a) its value; b) where it is stored; c) its scope. A variable whose value is an address in memory is called a *pointer*, and `*` is the *dereferencing operator* which passes from the value of a pointer to the value stored at that address. The first line of the body of `main` declares that `igl` is a pointer to an object of type `igList`. More literally, it says that the *pointee* `*igl` of `igl` is an `igList`. Up above we saw `char *argv[]`; this is a declaration (with highly abbreviated syntax) that `argv` is a pointer to an array of strings.

The *scope* of a variable is the portion of the code in which one can refer to it. If a variable, say `i`, is declared within the definition of a function, but not inside the definition of some smaller unit of code, then its scope is the definition of the function. When the execution of the program enters the definition of the function and then passes the variable's declaration, the variable comes to life, in the sense that it can be referred

to and memory is set aside for it. When the execution leaves the definition, the variable dies, and the memory set aside for it is freed. For most variables a rough rule of thumb is that the scope of a variable is the smallest portion of code of the form

```
{ ... }
```

that contains its declaration. Note that an `i` occurring in the definition of some other function is completely unrelated to the `i` used here, except that they happen to have the same name.

There is also a global storage class for variables that come into existence once and continue to exist throughout the execution of the program. The variables that `main` refers to that aren't declared within `main` are necessarily of this sort.

In C the syntax for invoking a function is

```
yourfunction(yourargumentlist);
```

or

```
yourreturnvalue = yourfunction(yourargumentlist);
```

(Again, we stress that while `yourreturnvalue` might be your primary interest, it also might not, and the consequences of invoking the function might go far beyond finding out what `yourreturnvalue` is.) Thus we see that the next line

```
igigSetup();
```

is telling `main` to run the procedure `igigSetup` which takes no inputs. Where is `main` supposed to learn about `igigSetup`? To figure this out we go back to the beginning of the file `demo1.c`. The way to add comments to C code is to put material (often on multiple lines) between the two character symbols `/*` and `*/`. At the beginning of `demo1.c` we find a bunch of boilerplate of this sort, that will be ignored when the executable is built, and then the lines

```
#include "igig.h"

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
```

These lines tell `demo1.c` where to look for the information it needs.

At this point we should say a bit more about the way a C program is broken down into files, and what happens to these files when the executable is built. In the directory `ig-1.1` do

```
> make clean
> ls
```

The program `make` is not "officially" part of C, but it provides the format for the vast majority of C programming. Basically it processes the `Makefile`. The command `make`

Figure 3: `complex.h`

```
typedef struct complex {
    double re;
    double im;
}:

complex add(complex x, complex y);
complex sub(complex x, complex y);
complex mul(complex x, complex y);
complex div(complex x, complex y);

double abs(complex x);
```

`clean` tells `make` to look inside the `Makefile` and follow the instructions there for "building" `clean`. What this actually means is to remove all the files that end with `~` (when the `emacs` editor opens `yourfile` it saves a copy as `yourfile~`, which is useful on occasion, but mostly something we have no interest in keeping around) all files that end with `.o`, which are called *objects*, and the various executables. This returns us to the state of the directory when you first entered it. In addition to the subdirectory `doc`, which contains documentation, there are now only files with the suffix `.h`, which are called *header* files, and files with the suffix `.c`.

Now do

```
> make demo1
```

As it goes along `make` prints what it is doing (and, sadly, sometimes some painful error messages) to the terminal. Without worrying about the details, what we see here are lines in which `gcc` (the GNU compiler for C and C++) is *compiling* the objects `ig.o`, `iglist.o`, `igmatrix.o`, `igtableau.o`, `igfp.o`, `iggsl.o`, and `igig.o`, and then a line in which these files and `demo1.c` are *linked* to build the executable `demo1`.

To get a better sense of what the two phases here are, let's consider a made up example. Suppose that the file `complex.h` is as shown in Figure 3. Here the `struct` construction declares that there is a structure called `complex` whose *members* are two `double`'s (`double` is one of C's builtin floating point types) and the preface `typedef` makes it the case that the complex numbers are, in certain subtle ways, treated as a new type. There are declarations of functions `add`, `sub`, `mul`, and `div` that each take two `complex`'s as arguments and return a `complex`, and a function `abs` that takes a `complex` as an argument and returns a `double`. But `complex.h` doesn't tell us how these functions are defined.

Suppose there is another file `moebius.c` that uses complex numbers to perform certain calculations. In the larger process of going from the source code to an executable, one can go a certain distance in processing `moebius.c` just using the information about complex numbers given by `complex.h`, and roughly speaking this is what happens when we compile `moebius.o`. Once we have compiled `complex.o` and `moebius.o`, we might link these with a file `toplevel.c` containing the definition of `main` to create the executable `toplevel`.

After this digression, let's go back to the definition of `main` in `demo1.c`. For the time being the main message is simply that variables or functions that are not declared in the body of `main` are declared earlier, either higher up in `demo1.c` or in some directly or indirectly included header file. In point of fact `igig.h` includes `iggsl.h`, which includes `igfp.h`, which includes `iglist.h` and `ig.h`. As we'll see in the next section, `igigSetup()` is declared in `igig.h`; it initializes various parameters of the function that searches for a fixed point.

The lines

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
```

include components of the Standard C Library, which is a rich resource containing tools for many common programming tasks. In particular, `stdio.h` declares functions that define "standard" input and output, including the function `printf` that you can see further down in the definition of `main`.

After the invokation of `igSetup()`, the next thing we see in `main` are four lines that assign values to variables using the assigment operator `=`. We set `igDim` to 4 because we are going to be dealing with a function from a four dimensional domain to itself, and we set `igigDmin` to 0.0 and `igigDmax` to 1.0. Obviously `igDim`, `igigDmin`, and `igigDmax` are global variables. The effect of these settings is that the initial phase of the process, in which random points are generated, these random points are drawn from the unit cube $[0,1]^4$. The subsequent search for a fixed point is *not* restricted to this cube unless, as in the function for this example, the function is defined in a way that forces the image of any point from the cube to again be in the cube. *It is possible to use the `IG` package to search for fixed points of functions from all of* $\mathbb{R}^{\texttt{igDim}}$ *to itself.* Of course such functions need not have fixed points, in which case it is possible for the search to "fail gracefully."

There is also a line

```
igFunction = *f1Apply;
```

says that `igFunction` is the pointee of `f1Apply`. Here we see that C has a construction called a *pointer to function*, that `igFunction` is a global variable whose values are functions, and that `f1Apply` is a pointer to a function. The pointer to function mechanism can be useful in other programs because it allows the decision about what to do (which function to invoke) to be decided on the basis of runtime information. For us its utility is that it permits the user to name the pointer to her function whatever she wants and then write one line of code that assigns its pointee to `igFunction`.

The next line of `main` sets `igl` (another global variable) to the return value of `igNewList` when it is invoked with argument `igDim`. Concretely, this sets up `igl` as a list of `igDim`-dimensional vectors that is initially of zero length.

The line

```
igfps = igigFindFP(igl, 0U);
```

Figure 4: `f1Apply`

```
igBOOL f1Apply(double *x, double *y)
  {
  if      (x[0] <= 0.0) { y[0] = 1.0; }
  else if (x[0] >= 1.0) { y[0] = 0.0; }
  else                  { y[0] = 1 - x[0]; }

  if      (x[1] <= 0.0) { y[1] = 1.0; }
  else if (x[1] >= 1.0) { y[1] = 0.0; }
  else                  { y[1] = (1 - x[1])/(1 + x[1]); }

  if      (x[2] <= 0.0) { y[2] = 1.0; }
  else if (x[2] >= 1.0) { y[2] = 0.0; }
  else                  { y[2] = sqrt(1 - x[2]*x[2]); }

  if      (x[3] <= 0.0) { y[3] = 1.0; }
  else if (x[3] >= 1.0) { y[3] = 0.0; }
  else                  { y[3] = 1 - sqrt(1 - (x[3]-1)*(x[3]-1)); }

  return igTRUE;
  }
```

is where the real work takes place. There is a search for a fixed point, and if one is found it is appended to `igl`. The argument `0U` ("zero you") is an argument that tells the function how much information about the computation to print to `stdout` ("standard out") as the function is computed. (There is a builtin type `unsigned int`, and `0U` denotes the unsigned version of zero.) At the conclusion of this process the return value of the function is assigned to `igfps`, which is a boolean value that is `true` if a fixed point was found and `false` otherwise.

Below that, there is a larger block of code that reports the results of the search, that is largely self explanatory, so we'll only comment on a few points. Note that the symbol `==` for a test of whether two variable have the same value is different from the assignment operator `=`. The `printf` function, from the Standard C Library `stdio`, is a bit of a world unto itself that one slowly gets used to, without ever fully losing the sense of ugliness that is such a strong first impression. Its first argument is a format, which often includes literal text, but also includes instructions for how to display the remaining arguments of the function. The arguments like `igl->dn[igl->size-1]` refer to various members of `*igl`, which is a structure of type `igList`.

Finally we have the lines

```
free(igl);
exit(EXIT_SUCCESS);
```

When a variable dies at the end of its scope, the memory that was set aside to hold its value is automatically freed for other uses, but with a pointer that may not be the end of the story, because we may (or may not) also cease to have any use for its pointee. If we also want to get rid of the pointee, we use `free` to do this. The last line tells the program to halt, returning the value of `EXIT_SUCCESS` to whatever process called `demo1`.

Let's next look at the definition of `f1Apply`, which is shown in Figure 4. It is not hard to figure out that this code embodies the composition of the function

$$(x_0, x_1, x_2, x_3) \mapsto (M(x_0), M(x_1), M(x_2), M(x_3))$$

(where M is the function $t \mapsto \max\{0, \min\{1, t\}\}$) with the function

$$(z_0, z_1, z_2, z_3) \mapsto \Big(1 - z_0, (1 - z_1)/(1 + z_1), \sqrt{1 - z_2^2}, \sqrt{1 - (z_3 - 1)^2}\Big).$$

Apparently the mathematical side of C is a bit more straightforward than the other parts we've seen.

Evidently the `IG` package is a rather complex beast, but for someone who already knows C it is possible to use it without knowing too much about what goes on inside. You will have to replace `f1Apply` with a function you're interested in, and you will have to write the code that uses the output as you wish. As with C itself, the best approach is to learn by doing, which means starting with smaller projects that develop and apply the most basic knowledge, then increasing your ambitions as you come to know more and more.

Here are some simple exercises to get you started. Copy `demo1.c` to `mydemo.c`. Edit the `Makefile` to make it the case that `mydemo` compiles along with everything else when you do `make` or `make all`, and by itself when you do `make mydemo`. Also, be sure that it gets cleaned up when you do `make clean`. Now edit `mydemo.c`, changing several things. First write a function

```
double Mm(double t) {
...
}
```

that computes the function M above. Then use this function to simplify the code for `f1Apply`. Now modify the definition of `f1Apply` to make it a bit more interesting, perhaps because it has multiple fixed points. Finally, reduce the number of digits that are printed to make the output prettier.

You should be able to do all of this without looking at books on `make` and C, simply by extrapolating from the patterns you see in the files. But as you go along, you must be sure to test what you have done after each step by compiling and running `mydemo`. Bugs creep into C programs during even the simplest edits, and the only way to suppress them is by constantly testing and fixing anything that goes wrong.

3 Theoretical Background

Let's fix a nonempty compact convex set $C \subset \mathbb{R}^d$ and a continuous function

$$f : C \to C.$$

In this section we describe a small part of the mathematical theory of computing fixed points of f, and how these ideas are incorporated in the `IG` package.

The first point, which is quite obvious, is that you cannot hope to reliably compute an exact fixed point of f, simply because the components of its fixed points may be transcendental numbers without any exact finite representation. A more subtle point is that it is also unreasonable to ask for a point that is certainly close to some actual fixed point of f, at least if the algorithm learns about f by sampling its values. To see why suppose that your algorithm had computed $y_1 = f(x_1), \ldots, y_k = f(x_k)$ and declared that x was within ε of some fixed point. The Devil could now change f by deforming C while holding $x_1, \ldots, x_k, y_1, \ldots, y_k$ fixed in a way that moved all the fixed points away from x. Your algorithm would treat the Devil's function the same way it treated your f, but the Devil's function doesn't have any fixed points near x. It seems that the best we can hope for is an *ε-approximate fixed point*, which is a point x such that $\|f(x) - x\| < \varepsilon$.

3.1 Methods that Might Converge

Let's begin by considering some obvious, rather naive ways, to search for ε-approximate fixed points. You could use a random number generator to generate random points in C until you found one that happened to be an ε-approximate fixed point. This might take a *very* long time.

You could compute a sequence $x_0, x_1, x_2, \ldots$ by choosing x_0 randomly and letting $x_i = f(x_{i-1})$ for each $i = 1, 2, \ldots$. This is called *function iteration*. With lots of examples it succeeds, but there are many other, by no means pathological, examples where the sequence never arrives at an ε-approximate fixed point.

If f is sufficiently smooth there are other things you can try. Let $g : C \to \mathbb{R}^d$ be the function $g(x) = f(x) - x$. Finding a fixed point of f is the same thing as finding a zero of g, and we can apply Newton's method to the latter problem. Specifically, we could choose x_0 randomly and compute a sequence $x_1, x_2, \ldots$ by letting

$$x_{i+1} = x_i - Dg(x_i)^{-1} g(x_i).$$

Sometimes this will converge to a fixed point of f, but sometimes the sequence will cycle forever or eventually leave C.

This procedure can be modified by letting

$$x_{i+1} = x_i - \delta_i Dg(x_i)^{-1} g(x_i)$$

where δ_i is some small positive number that may depend on x_i in various ways. In the limit as δ_i becomes small this process approximates following the flow of the vector field given by the negative of the gradient of the function $x \mapsto \|g(x)\|$. In effect, the process

slides downhill until it can't go any further. Once again, this process might well find a fixed point, but it can also end at a local minimum of this function that is not a zero, or even at a critical point that is not a minimum.

An important computational point is that these methods don't *actually* require that f be differentiable. For $i = 1, \ldots, d$ let $\mathbf{e}_i = (0, \ldots, 1, \ldots, 0)$ be the i^{th} standard unit basis vector of $\mathbb{R}^d$. For some small positive number δ we can compute a matrix $\Delta g(x)$ by letting $\Delta_{ij} g(x)$ be the i^{th} component of $\delta^{-1}\big(g(x + \delta \mathbf{e}_j) - g(x)\big)$. This could fail to be a good proxy for $Dg(x)$ for any number of reasons, but nothing else we have been talking about so far comes with any guarantees, and it can always be computed, at least if x is not on the boundary of C. In many circumstances it will work quite well.

3.2 Methods that Should Converge

There are methods that come with a theoretical guarantee that a fixed point will be found except in degenerate cases. From a practical point of view the most important of these is homotopy. Let x_0 be a point in C, and let $h : C \times [0, 1] \to C$ be the function

$$h(x, t) = (1 - t)x_0 + tf(x).$$

It is customary to think of t as time, to let h_t denote the function $h(\cdot, t) : C \to C$ at time t, and to say that h is a *homotopy* connecting the constant function h_0 and $h_1 = f$. If f is smooth, then it is almost always (in a sense that can be made mathematically precise) the case that the connected component of

$$Z = \{\, (x, t) : h(x, t) = x \,\}$$

that contains $(x_0, 0)$ is a path whose other endpoint has the form $(x_1, 1)$, so that x_1 is a fixed point of f.

Following this path is simple in theory, but a bit tricky in computational practice. The idea is to walk along the path by repeatedly combining a *predictor* step, that uses the derivative of h (or a difference quotient proxy for it) to guess where the point a certain distance down the path will be, with a *corrector* step that uses Newton's method to pass from the guess to an actual point on the path. If one tries to take too big a step it can happen that Newton's method diverges off into the wilderness. This actually isn't so bad, because this situation can be diagnosed, and one can go back and try a more modest step size. The really scary possibility is that, without realizing it, the procedure could "hop" from one component of Z to another. If the new component is a loop, which is possible, then the procedure might just go around it again and again. There are assumptions that guarantee that such hops won't happen if the step size is sufficiently small, but it is hard to know in advance what "sufficiently small" means here. One can try to be quite conservative when guessing a step size, but then it can happen that the path is quite long and winding. Actually, the path can be long and winding just because it is, without any reference to the step size. In spite of all this, the homotopy method is quite practical for many application domains, and the HOMPACK package continues to find applications even though it has been around for many years.

We should also mention the Scarf algorithm. In this algorithm C is a simplex that has been *triangulated*, which means that it has been subdivided into smaller simplices.

The procedure goes along a path of adjacent simplices, starting from a known initial point, until it finds one that satisfies a condition that is not quite the same as being an ε-approximate fixed point, but is nonetheless a reasonable proxy for approximate fixedness. As with homotopy, the path can be long and winding, and progress through C can be quite slow, especially when the dimension d is large. Although it remains quite important in theoretical work, the Scarf algorithm has not turned out to be very useful in computational practice.

3.3 The Imitation Game Algorithm

We now describe the method developed in [9]. To begin with, let's imagine a game with two players who simultaneously choose points from C. Player 1 wants to choose a point that is as close as possible to the image of Player 2's choice under f, and Player 2 wants to choose a point that is as close as possible to Player 1's choice. In order for a pair (x, y) to be a pure Nash equilibrium of this game it must be the case the $x = f(y)$, since otherwise Player 1 could improve her choice, and that $y = x$, since otherwise Player 2 could improve her choice. That is, if (x, y) is a pure Nash equilibrium, then $y = x = f(y)$. Conversely, if x is a fixed point of f, then (x, x) is obviously a pure Nash equilibrium.

We are now going to consider a finite version of this game. Let $x_1, \ldots, x_m$ be points in C, and let $y_1 = f(x_1), \ldots, y_m = f(x_m)$. We will think of the strategy set of both agents as $\{x_1, \ldots, x_m\}$. We adopt the following payoff functions for the two players: if Player 1 chooses x_i and Player 2 chooses x_j, then their payoffs are

$$a_{ij} = -\|y_j - x_i\|^2 \quad \text{and} \quad b_{ij} = \delta_{ij}.$$

(Here δ_{ij} is the Kronecker symbol: $\delta_{ij} = 1$ if $i = j$ and otherwise $\delta_{ij} = 0$.) That is, Player 1 wants to minimize the square of the distance between her choice and the image of Player 2's choice, and Player 2 wins if and only if her choice is the same as Player 1's choice. A game in which the two agents have the same set of pure strategies, and the second player's payoff matrix is the identity matrix, is an *imitation game*.

A mixed Nash equilibrium is a pair whose components are mixed strategies for the two players—that is, probability distributions on $\{x_1, \ldots, x_m\}$—such that each player is maximizing her expected utility if she takes the other player's mixed strategy as given. A player's expected utility is the sum over her pure strategies of the probability that she plays that pure strategy times her expected utility if she does, so a pair of mixed strategies is a Nash equilibrium if and only if each player is assigning all probability to pure strategies that maximize expected utility, taking the other player's mixed strategy as given. What this means for Player 2 is obvious: *Player 2 is maximizing expected utility if and only if she is assigning all probability to the x_i that Player 1 chooses most frequently.*

What equilibrium means for Player 1 emerges from a brief calculation. Suppose that $\tau = (\tau_1, \ldots, \tau_m)$ is Player 2's mixed strategy, and let

$$z = \sum_{j=1}^{m} \tau_j y_j.$$

If Player 1 plays x_i, then her expected utility is

$$\begin{aligned}
-\sum_{j=1}^{m}\tau_j\|x_i-y_j\|^2 &= -\sum_{j=1}^{m}\tau_j\big\langle (x_i-z)+(z-y_j),(x_i-z)+(z-y_j)\big\rangle \\
&= -\sum_{j=1}^{m}\tau_j\big\langle x_i-z,x_i-z\big\rangle \\
&\qquad -2\sum_{j=1}^{m}\tau_j\big\langle x_i-z,z-y_j\big\rangle \\
&\qquad\qquad -\sum_{j=1}^{m}\tau_j\big\langle z-y_j,z-y_j\big\rangle \\
&= -\|x_i-z\|^2-\sum_{j=1}^{m}\tau_j\|z-y_j\|^2
\end{aligned}$$

because $\sum_{j=1}^{m}\tau_j=1$ and

$$\sum_{j=1}^{m}\tau_j\big\langle x_i-z,z-y_j\big\rangle = \big\langle x_i-z,z-\sum_{j=1}^{m}\tau_j y_j\big\rangle = \big\langle x_i-z,0\big\rangle = 0.$$

Since the final expression is the sum of $-\|x_i-z\|^2$ and a term that does not depend on i, we see that *Player 1 is maximizing her expected utility if and only if she is assigning all probability to the x_i that are closest to z.*

We can now let $x_{m+1}=z$ and $y_{m+1}=f(z)$, and we are back where we started with m replaced by $m+1$. Choosing x_1 randomly, setting $y_1=f(x_1)$, then repeating this process again and again generates a sequence $\{x_m\}$ in C. Since C is compact, this sequence necessarily has an accumulation point, say x^*, and we claim that x^* must be a fixed point of f. If not, then we can choose disjoint neighborhoods U of x^* and V of $f(x^*)$ with V convex and $f(U)\subset V$. Every neighborhood of x^* contains infinitely many terms of the sequence, so there is an $x_m\in U$ that comes from an equilibrium of the game in which Player 1 assigns all probability to points in U, because the points in $\{\,x_n : n<m\,\}$ that are closest to x_m are all in U, and consequently x_m is a convex combination of the images of such points, so it is in V. Since U and V are disjoint, this is impossible.

Not only is x^* a fixed point, but *the sequence $\{x_m\}$ contains an ε-approximate fixed point* because the set of such points is a neighborhood of x^* and the sequence eventually gets inside any neighborhood. Thus we have described a procedure that is guaranteed to eventually find an ε-approximate fixed point. But unlike homotopy and the Scarf algorithm, it is can maintain this guarantee without clinging tightly to a possibly quite lengthy one dimensional object. Instead, the guarantee comes from the fact that it cannot return again and again to the vicinity of any point that is not fixed. Our intuitive feeling is that it while it shares the main advantage of homotopy and the Scarf algorithm, it should also be agile, often quickly zooming in to the neighborhood of a fixed point after a small amount of experimentation.

Providing theoretical tools for analyzing the relative merits of different algorithms for finding approximate fixed points is a major theoretical challenge. In [6] Hirsch, Papadimitriou, and Vavasis showed that all algorithms for finding fixed points that obtain

their information about the function by sampling its values have exponential worst-case complexity. Once you commit to an algorithm, the Devil can concoct a function that will stymie it. In [5] Goldberg, Papadimitriou, and Savani show that certain homotopy algorithms are "complete" for the computational class **PSPACE**, which is large and is thought to contain very hard problems. What this means concretely is that one can pass in polynomial time from any problem in **PSPACE** to a "rendering" of it as a fixed point problem, such that if one of the homotopy algorithms they consider is applied, it terminates at a point that can easily (that is, in polynomial time) be interpreted as a solution of the given problem from **PSPACE**. This gives a precise sense in which homotopy solvers are solving a harder problem than is really necessary. However, this complexity result is out of line with the practical success of homotopy software in practice. One would like to approach the issue from an average-case or typical-case perspective, but even for very specific application domains there do not seem to be any obvious candidates for "natural" distributions over problem instances.

Since the imitation game solver needs to find a Nash equilibrium of a two player game, we should also say a bit about the theory related to this problem, which is quite interesting. For a long time, whether there was a polynomial time algorithm for this problem was a prominent open problem. In [12] Savani and von Stengel showed that the Lemke-Howson algorithm has an exponential worst case running time. Around the same time [3] and [1] showed that this problem is "complete" for the class of computational problems **PPAD**. Since PPAD contains problems that are thought to be very difficult, it now seems extremely unlikely that there is a polynomial time algorithm for finding Nash equilibria, or even a subexponential algorithm.

How good is our procedure in practice? Preliminary experience suggests that it is very good, particularly for high dimensional problems. But one must bear in mind that the answer to this question may vary from one application domain to the next. In addition, it is more a matter of engineering than of theory, because our software is not a "pure" rendition of the procedure, but instead takes advantage of:

- random search to find a starting point that seems propitious, due to the relatively small distance from it to its image, and
- the Newton method, to quickly improve the quality of the approximation once we are in the neighborhood of a fixed point.

We use the Lemke-Howson algorithm to find Nash equilibria of the imitation games. This has one potentially quite important advantage, namely that the initial pivot can be chosen to be close to the point x_m most recently added to the set. But any two player game solver could be substituted for Lemke-Howson, either all the time or perhaps in connection with games where Lemke-Howson doesn't seem to be working. More generally, the problem of finding a Nash equilibrium has a representation as a fixed point problem, so we can apply any fixed point finder. In particular, there is the seemingly whimsical possibility of recursively applying the imitation game algorithm itself to perform this step of the computation!

4 Code

Of the various resources that exist in the `IG` package, some are visible in the various header files, and are therefore directly accessible to the example programs and software you might write, while others lurk in the bowels of the `.c` files. The latter can be made public by putting their declarations in the relevant header file, so they are not inaccessible, but as the code stands now they are regarded as implementation details that the user need not be concerned with.

A next step in the process of familiarizing the reader with the `IG` package is to go through the `.h` files, listing the things they contain, and briefly explaining how they work. The first parts of the following descriptions of the header files are in that spirit, with the portions labelled **The Nitty Gritty** discussing implementation details that are mostly related to the corresponding `.c` files.

4.1 `ig.h`

This part of the code defines elementary types and utilities. To begin with, as is customary in C programming, we create a Boolean type: an `igBOOL` is a Boolean variable that takes on the values `igTrue` and `igFalse`, which are just fancy symbols for 1 and 0 respectively. The function

```
igAbend(char *file, int line, char *msg);
```

is used to terminate the program under abnormal circumstances, sending an error message with the line number of the file in which the program terminated to `stderr` ("standard error"). This is primarily for debugging, and if everything is working properly it should never be invoked.

There is a global variable

```
igRandomSeed
```

that is an `unsigned int`. To understand this you need to know how the random number generator given by the Standard C Library works. The random number generator has an internal state, and a call to the function `rand()` deterministically returns a pseudorandom number and updates the seed, so that repeated calls to `rand()` give a pseudorandom sequence of numbers. The user can set the internal state by invoking the library function `srand(s)` whose argument `s` (the "seed") is an `unsigned int`. By using `srand(s)` at the beginning of the run, one can insure that the same pseudorandom sequence will be produced every time, which can be useful for debugging purposes if, for instance, one notices a problem.

There is a function

```
void igRandSeed(unsigned int s);
```

This function first checks whether `s` happens to be `0U` (recall that this is unsigned zero) in which case `s` is reassigned to a value generated by using some library functions that get

data (calendar time, the amount of time elapsed since the program started, the process id) that is random from the perspective of the run of the program. Then `igRandSeed` calls `srand(s)` and sets `igRandomSeed` equal to `s`. During programming and debugging this feature can be used to make the seed "truly random" most of the time while keeping track of `igRandomSeed` in order to be able to repeat some run that was, for some reason, interesting.

The pseudorandom numbers, which are uniformly distributed, are used by the following function to produce normally distributed pseudorandom numbers, using the Box-Muller method (see, e.g., [11, Section 7.2]):

```
double igNormal(double mean, double sd);
```

There is now a collection of utilities for producing and manipulating vectors. We can produce vectors with independent normally and uniformly distributed components. In addition there are functions that produce vectors on the diagonal, and vectors with uninitialized components.

```
double* igGetNVect(int dim, double mean, double sd);
double* igGetUVect(int dim, double min, double max);
double* igGetCVect(int dim, double c);
double* igGetVect(int dim);
```

We can read a vector from `stdin` (standard input) and produce a copy of a given vector.

```
double* igReadVect(int dim);
double* igCopyVect(int dim, double *v);
```

The following functions give $\|v_1 - v_2\|_1$, $\|v_1 - v_2\|_2$, $\|v_1 - v_2\|_\infty$, respectively:

```
double igL1Norm(int dim, double *v1, double *v2);
double igL2Norm(int dim, double *v1, double *v2);
double igLinfNorm(int dim, double *v1, double *v2);
```

There are functions that test, respectively, whether all the components of `v` are finite and whether one or more components is within `tol` of `min` or `max`:

```
igBOOL igVectFinite(int dim, double *v);
igBOOL igOnBound(int dim, double *v, double min, double max, double tol);
```

The function `igClipVect` moves each component to the nearest point in the interval [`min`, `max`]. It returns `igTrue` if and only if any coordinate was modified.

```
igBOOL igClipVect(int dim, double *v, double min, double max);
```

Finally, there is a function, used primarily in debugging, that prints a vector after a string `s` that is a user-supplied message of whatever sort:

```
void igPrettyP(int dim, double *v, char *s);
```

The Nitty Gritty: Note that all these `ig*Vect()` functions allocate memory, and it is the user's responsibility to free this when it is no longer required. Care is taken in the `IG` package to ensure that all the `double` values used are valid floating-point numbers, where valid means: is finite and is not a "not a number" value. The `isfinite()` system library call is used for this, and the `igVectFinite()` function is provided to check that all the coordinates of a point are valid.

4.2 `iglist.h`

The `IG` package keeps track of the fruits of its labor using a data type `igList`. An `igList` can be thought of as a list of

$$\text{(domain point, image point, distance between the two)}$$

triples. The type specification has a bit more than this:

```
typedef struct
  {
  int dim;              /* dimension of the point data */
  int alloc, size;      /* allocated & used space */
  double **dx;          /* the X point data */
  double **dy;          /* the Y point data */
  double *dn;           /* the norm data */
  }
igList;
```

Here `dx` and `dy` are lists of `dim`-dimensional vectors, and `dn` is the list of distances between the vectors in `dx` and `dy`. The member `size` is the length of the lists, and `alloc` is used in the memory management tasks related to creating, updating, and freeing an `igList`. The following functions initialize a pointer to `igList`, release the memory devoted to its pointee, shrink the pointee by discarding all but the first `s` items on the lists, and expand it by appending an item to the list.

```
igList* igNewList(int dim);
void igFreeList(igList *igl);
void igShrinkList(igList *igl, int s);
void igAppend(igList *igl, double *x, double *y, double n);
```

The Nitty Gritty: An `igList` object tracks the amount of space allocated by the system for its own lists and the amount actually used. Space is allocated in 'chunks' (set by the `CHUNK` defined constant in `iglist.c`, currently 16 entries), with new space being allocated using the `realloc()` system call to preserve any existing data.

An `igList` structure is created by the `igNewList()` function, with initial allocated space of one chunk and used space of zero. When a new tuple is appended to an `igList` by `igAppend()` function, if the amount of allocated space is not sufficient for the new list size, another chunk is automatically allocated. Note that the X and Y data are passed in and stored as pointers, and that the coordinate data is not copied. The `igShrinkList()`

function shrinks the number of tuples in an `igList` object to the number given as an argument by removing the requisite number of items from the end of the list and freeing any space allocated to them, with the allocated space for the `igList` object itself remaining the same. The memory used by an `igList` structure and its contents should be freed when no longer required by the `igFreeList()` function. Note that freeing a list also frees the data arrays associated with each of the X and Y points.

Note that `igNewList()` takes a dimension as an argument. This value is stored but not used. In particular, there are no checks to ensure that the dimensions of any appended X and Y points match this value. Also, there are no checks to ensure that the coordinates of these X and Y points are 'finite' (not `INF` or `NAN`)

4.3 `igmatrix.h`

There is a square matrix type:

```
typedef struct
  {
  int alloc, size;     /* allocated & used space */
  double **dpp;        /* data */
  double min, max;     /* running min/max values */
  igBOOL mset;         /* min & max have been init'd */
  }
igMatrix;
```

An `igMatrix` is a `size` $\times$ `size` array of doubles. As with `igList`, `alloc` is a variable used in the memory management tasks. The members `min` and `max` are in principle the values of the maximum and minimum entries. The functions

```
igMatrix* igNewMatrix(void);
void igFreeMatrix(igMatrix *igm);
```

respectively create a new `igMatrix` and return a pointer to it, and destroy the pointee of a pointer to an `igMatrix` by freeing the memory assigned to it. The functions

```
void igResize(igMatrix *igm);
void igSetIJ(igMatrix *igm, int i, int j, double d);
```

respectively expand the matrix by adding a row and column, without initializing their entries, and set the value of an entry, decreasing `min` or increasing `max` if `d` is outside the previous bounds.

The Nitty Gritty: The structure keeps track of the amount of space allocated by the system for the array and the amount actually used. Space is allocated in 'chunks' (set by the `CHUNK` defined constant in `igmatrix.c`, currently 16 rows/columns), with new space being allocated using the `realloc()` system call to preserve any existing data. An `igMatrix` structure is created by the `igNewMatrix()` function, with initial allocated space of one chunk and used space of zero. The `igResize()` method increases the used space by

one row and one column. If the amount of allocated space is not sufficient for the new size, another chunk is automatically allocated. The memory used by an `igMatrix` structure and its contents should be freed when no longer required by the `igFreeMatrix()` function.

The data in the new row and column should be set using the `igSetIJ()` function in order to keep track of the maximum and minimum entries in the array. (The minimum value is needed to correctly set up the tableau for the Lemke-Howson algorithm.) Each entry of the matrix must be explicitly set, and this should be done once only, so `igSetIJ` should only be used to set new values of the matrix after it has been expanded by adding a row and column, and not to modify the values of entries that have already been initialized. Thus, for example, `igSetIJ` does not check to see if `min` has increased or `max` has decreased because the operation modified an entry that had previously determined one of these.

The `igMatrix` structure is specifically designed for use with the imitation game algorithm and the Lemke-Howson code, and is not suitable for general use. There is no support for reusing such a structure or resetting its size or contents. It should not be necessary for users of the `IG` package to create or manipulate `igMatrix` structures.

4.4 `igtableau.h`

The Lemke-Howson algorithm finds Nash equilibria for 2-person normal form games by solving the associated linear complementarity problem. The data for this problem is maintained in a "tableau," and the problem is solved by a series of pivoting steps (see, e.g., [7]). The cast of characters is as follows.

```
typedef struct
  {
  double ***tab;     /* the 2 sub-tableaux (2D arrays) for the X & Y players */
  int dim1, dim2;    /* the number of strategies of players 1 & 2 (ie, X & Y) */
  int *ind;          /* array for lexico-maximal check */
  int numP;          /* number of pivot operations */
  double zero;       /* the zero tolerance to use */
  }
igTableau;
```

There are subtableaus for the two player, so `tab` is an array whose entries are two matrices, each of which is in turn an array of arrays of doubles. Geometrically, pivoting is a matter of going along an edge of a polytope until one bumps into a new facet. In degenerate cases one can simultaneously bump into several facets, and one needs a rule for deciding which one is "really" (in a rather virtual sense) the one the algorithm collided with. The array of integers `ind` is used to keep track of the information required to compute the rule. As the algorithm proceeds, round-off error can accumulate, eventually corrupting the calculations, and for this reason one may wish to bound the number `numP` of pivots, or one may be interested in this number for other reasons. Round-off error can also preclude an exact solution, and `zero` is the required quality of approximation.

There are a number of bad things and one good thing that might happen to the tableau, that are encoded as follows:

```
typedef enum
  {
  igNotInit,     /* tableau not in initial state */
  igBadArg,      /* bad initial pivot or pivot limit argument */
  igPivLim,      /* pivot limit exceeded */
  igNoRow,       /* can't find positive coefficient row */
  igLexMax,      /* can't find lexico-maximal row */
  igNE           /* reached a Nash equilibrium */
  }
igLHSTAT;
```

The `igLemHow()` function implements the Lemke-Howson algorithm.

```
igLHSTAT igLemHow(igTableau *igt, int initP, int limP);
```

It takes as arguments:

- `igt`, a pointer to an `igTableau` structure representing the tableau for the game;
- `initP`, the strategy to use as the initial pivot;
- `limP`, the limit on the number of pivoting steps.

The return value of `igLemHow()` indicates which of the conditions above caused termination.

Of course if the algorithm succeeds one will typically want to get the Nash equilibrium mixed strategies, and for various reasons one may wish to know the number `numP` of times that the algorithm pivoted.

```
double* igGetXProb(igTableau *igt);
double* igGetYProb(igTableau *igt);
int igGetPivots(igTableau *igt);
```

The larger algorithm needs to be able to create tableaus that are used to solve imitation games.

```
igTableau* igNewTableau(igMatrix *igm, double tol);
```

The `igNewTableau()` function allocates the space for an `igTableau` structure and initialises it with the data from an `igMatrix` structure to represent the imitation game. Note that the `igMatrix` data, which represent the payoffs to the first player, are translated so that they are all positive (with a minimum of 1.0). The initial tableau created by `igNewTableau()` represents the null equilibrium (aka, the artificial or extraneous equilibrium) of the imitation game.

It is the user's responsibility to free the memory used by an `igTableau` structure when no longer required by the `igFreeTableau()` function.

```
void igFreeTableau(igTableau *igt);
```

For debugging purposes there is a function that prints a tableau.

```
void igDumpTab(igTableau *igt);
```

The Nitty Gritty: If the first and second players have, respectively, d_1 and d_2 strategies, then these are labelled 1 to d_1 and $d_1 + 1$ to $d_1 + d_2$. The corresponding slack variables are labelled with the negatives of these. The C arrays are indexed from 0, with the first column of the tableau giving the current basis (as a mix of strategy and slack labels) and the second column giving the basis weights. The private functions `getPivot()`, `getTableau()` and `getColumn()` handle the details of translating variable and slack labels into the appropriate pivot strategy, sub-tableau and column.

The zero tolerance in the `igTableau` structure is used in the `igLemHow()` function to establish whether or not floating-point values should be treated as zero, but this is not sufficient to eliminate all floating-point problems. Note that numerical problems mean that, even though the tableau represents a non-null Nash equilibrium, all the probabilities in one of the vectors extracted by `igGetXProb()` and `igGetYProb()` can be zero; this is represented by a return value of `NULL`.

The `IG` package uses `igLemHow()` only on imitation games and makes a single call to `igLemHow()` for each newly constructed null equilibrium tableau, with the initial pivot being one of the first player's strategies. The details of this are handled by the `igImitate()` function, and there is no need for the user of the `IG` package to directly create or manipulate `igTableau` structures.

The Lemke-Howson algorithm itself, and so the `igLemHow()` function, can solve arbitrary (including non-square) games, need not be started at the null equilibrium, and can use any choice of initial pivot. But `igNewTableau()` is the only currently existing function for initializing a tableau, so a user who wishes to take advantage of this generality must write new initialization functions. Since the `igLemHow()` routine is written for general-purpose use, its running time when used by the `IG` package could, in principle, be improved by taking into account the special form of the imitation games employed. In particular, the payoff matrix for the second player is the identity matrix. Thus every second pivot step is essentially vacuous, and could be avoided. In practice the speed-up is likely to be minimal, since the pivoting code is written so that rows which have zero (in the relevant column) are skipped during pivoting.

The `igLemHow()` function may fail if the tableau is not in its initial state when the function is called (this is tracked by the `numP` structure member). That is, `igLemHow()` should only be called once on an `igTableau` after it is created.

The files `igtableau.h` and `igtableau.c` were created by concatenating a set of source files which implemented a stand-alone version of the Lemke-Howson algorithm and a 'lookalike' of Gambit's `gambit-lcp` command line tool (see [8]). This code was based on the original code by Codenotti et al. [2] (which is dated August 2006), as modified to include the lexico-minimal ratio test to cope with degenerate games (see, for example, [7]). The files were then edited to excise unused code, to add some new code to trap various floating-point induced errors in the `igLemHow()` function, and to integrate them into the `IG` package.

4.5 igfp.h

We are now at the point where fixed points will be computed. There are the following global variables.

```
extern int igDim;        /* dimension of X & Y spaces */
extern double igTol;     /* tolerance on the X-Y norms */
extern double igZero;    /* tolerance on zero values or differences */
extern int igPLimit;     /* limit on number of Lemke-Howson pivots */
```

Here `igTol` is the largest distance from the domain point to the image point that is acceptable in order for the domain point to be regarded as an approximate fixed point, and `igZero` is the tolerance of inexactitude in the Lemke-Howson algorithm.

In the Lemke-Howson algorithm any of the pure strategies, for either player, can be the one whose equilibrium condition is relaxed. Intuitively one should expect that the Lemke-Howson algorithm will, on average, find an equilibrium faster if it begins with a pure strategy that corresponds to a point in the domain of the function that is, in some sense, already a pretty good guess about where the approximate fixed point will eventually be found. We provide labels for three different ways of choosing this point:

```
typedef enum
  {
  igLHlast,    /* use the last (most recent) X on the list as initial pivot */
  igLHnorm,    /* use X from list with min'm X-Y norm as initial pivot */
  igLHbest     /* use strategy yielding best oldX-newY norm as initial pivot */
  }
igLHPIV;
```

The one in effect is a global variable:

```
extern igLHPIV igLHPivot;
```

All the functions that a pointer to function can point to must have the same return type and argument list.

```
typedef igBOOL (*igFUN)(double *x, double *y);
extern igFUN igFunction;
```

Thus an `igFUN` is a pointer to a function that takes two arrays of doubles (the domain point and its image) as arguments and returns an `igBOOL`. (Typically the return value indicates whether the attempt to compute `*y` succeeded.) There is a global variable `igFunction` that points to the function under consideration.

Similarly, all of the norms introduced in `ig.h` have the same "signature," so we can define a type that points to such functions and declare a global variable indicating which we are using.

```
typedef double (*igNORM)(int d, double *x, double *y);
extern igNORM igNorm;
```

There are three functions that compute approximate fixed points of the pointee of `igFunction`.

```
igFPSTAT igGuess(igList *igl, int cnt, double rmin, double rmax);
igFPSTAT igIterate(igList *igl, int cnt);
igFPSTAT igImitate(igList *igl, int cnt, unsigned int debug);
```

These finders all take an `igList` structure as their argument, use any data on this list as their starting point, and append to the list any result tuples $\langle X, Y, \|X - Y\|\rangle$. The tuples as checked as they are generated by the finders to ensure that all the coordinates are finite, that $f(X) = Y$, and that the norm is finite.

The function `igGuess()` generates `cnt` uniformly distributed random points X_i in the `igDim`-dimensional cube of points whose components are all between `rmin` and `rmax` for each $i = 1, \ldots,$ `cnt`. For each i it computes the value Y_i of the function at X_i, then the norm $\|X_i - Y_i\|$, and it appends the triple $(X_i, Y_i, \|X_i - Y_i\|)$ for which $\|X_i - Y_i\|$ is minimal to `*igl`. Function application is, in general, much faster than solving large imitation games or polishing approximate fixed points in high-dimensional spaces, so it is often feasable to use `igGuess()` with very large values for the number of guesses. The intended use of `igGuess()` is to generate initial X points for, e.g., the `igIterate()` and `igImitate()` functions.

Starting with the last triple $(X_0, Y_0, \|X_0 - Y_0\|)$ on the input list `*igl`, the function `igIterate()` computes a sequence

$$(X_1, Y_1, \|X_1 - Y_1\|), \ldots, (X_{i^*}, Y_{i^*}, \|X_{i^*} - Y_{i^*}\|)$$

where $X_1 = Y_0$, each Y_i is the value of the function at X_i, and $X_i = Y_{i-1}$ for $i = 2, \ldots, i^*$, and $(X_{i^*}, Y_{i^*}, \|X_{i^*} - Y_{i^*}\|)$ is appended to the list. Here i^* is either the first index not greater than `cnt` such that X_{i^*} is a within-tolerance fixed point, in which case a status of `igFP` is returned, or `cnt` if no such fixed point in found, in which case the return value is `igOK`. Note that this tuple is the most recently generated one, not necessarily the best-approximating one. The function fails, with the list unchanged and return value `igFail`, if the length of `*igl` is zero or there is a numerical problem.

Given a non-empty `igList`, the `igImitate()` function builds and solves the imitation game represented by the tuples on the list and then generates an $\langle X, Y, \|X - Y\|\rangle$ tuple from the solution (this is one iteration of the "while" loop in Figure 5). This tuple is appended to the list and the process repeated until we find a within-tolerance fixed point, `cnt` games have been built and solved, or there's an error. The call fails if no tuples are added to the list, while if a fixed point is found it will be the last item on the list. Note that all the valid tuples generated are appended to the list, since they're needed to build the imitation games. Thus, if no fixed point is found, the updated contents of the list need to be examined to establish the best approximation found.

If the input `igList` contains a single tuple, then building and solving the initial imitation game is equivalent to function iteration. This is not necessarily true for subsequent imitation games. If the initial list size is n and the function doesn't return until

Figure 5: The imitation game algorithm

Inputs:

d	the dimension of the space, $d \geqslant 1$
f	the function, $f : \mathbb{R}^d \to \mathbb{R}^d$
X_0	the starting point, $X_0 \in \mathbb{R}^d$
ϵ	the norm tolerance, $\epsilon > 0.0$
n	the norm to use, $n \in \{1, 2, \infty\}$
l	the limit on the game size, $l \geqslant 2$
p	the limit on the number of pivots, $p \geqslant 2$

Output:

an X such that $||X - f(X)||_n < \epsilon$, or a failure message

Algorithm:

set $Y_0 = f(X_0)$
if $||X_0 - Y_0||_n < \epsilon$, then return X_0 and stop
set $X_1 = Y_0$ and $Y_1 = f(X_1)$
if $||X_1 - Y_1||_n < \epsilon$, then return X_1 and stop
set $i = 2$
while $i \leqslant l$
    construct the $i \times i$ imitation game $G = (A, I_i)$, where $a_{rs} = -||X_{r-1} - Y_{s-1}||^2$
    set $P = \text{LemkeHowson}(G, p, X_{i-1})$, where X_{i-1} represents the initial pivot
        and P is player two's strategy from the Nash equilibrium
    set $X_i = \sum_{j=1}^{i} P[j]\, Y_{j-1}$
    set $Y_i = f(X_i)$
    if $||X_i - Y_i||_n < \epsilon$, then return X_i and stop
    set $i = i + 1$
return failure and stop

all `cnt` imitation games are built and solved, then the largest imitation game generated has size $n +$ `cnt` $- 1$ and `cnt` tuples are appended to the list. Note that if `igGuess()` is called n times with the same list as its argument it will populate the list with (up to) n approximate fixed points. These could then, e.g., be passed to the `igImitate()` function which would then start with an $n \times n$ imitation game instead of a game of size 1×1.

As we stressed in Section 2, `igImitate()` does not impose or depend on any bounds on the points that arise in the process of building the imitation games, so it can be applied to unbounded problems. Such bounds are only used in our "standard" method of generating an input list for `igImitate()`. Various functions discussed earlier provide other ways of building such lists.

The Nitty Gritty: The 'global' parameters which set the problem dimension (`igDim`), the tolerance to use when assessing $X - Y$ norms (`igTol`), the tolerance to use when assessing `double` values (`igZero`), and the limit on the number of pivot operations when performing Lemke-Howson (`igPLimit`) are declared in `igfp.h` and defined in `igfp.c`. Similarly for the choice of initial pivot to use when performing Lemke-Howson (`igLHPivot`,

of type `igLHPIV`), the pointer to the user's function (`igFunction`, of type `igFUN`), and the pointer to the norm function (`igNorm`, of type `igNORM`).

The `igImitate()` function starts by initialising many of its local variables to 'sentinal' values. These are important to guide the control flow of the program and to ensure that all unneeded allocated memory is freed before the routine returns. In particular, the `stat` variable is used to break out of the main loop if there's an error or if a fixed point is found, with its value indicating which event occurred.

The function next checks its arguments and then creates an `igMatrix` structure and populates it with the $-\|X_i - Y_j\|^2$ ℓ_2-norms derived from the input `igList`. The `cindex` variable is set to reflect the current size of the `igList` and `igMatrix`, and the main loop is entered. On all but the first iteration of the loop the last action of the previous iteration was to append a new tuple to the `igList`, so `cindex` needs to be incremented and the `igMatrix` updated with the new information.

The function now builds an `igTableau` from the `igMatrix`, finds a Nash equilibrium using the Lemke-Howson algorithm, and uses the second player's strategy probability vector to build the next X point. The pivot to use when running the Lemke-Howson algorithm is determined by the `IG` package's `igLHPivot` parameter. If this is `igLHlast` it's the most recent X point on the list (i.e., the last), as in Figure 5. If it's `igLHnorm` it's the X point on the list whose tuple has the smallest $\|X - Y\|$ norm.

The algorithm in Figure 5 has a performance guarantee, in the sense that the sequence of X_i it generates is guaranteed to contain a convergent subsequence if the function satisfies certain conditions. So, the normal pivot choice is `igLHlast`. If the pivot parameter is `igLHbest` then all pivot choices are tested and the one which yields the next X point which is closest to the most recent X point on the list is used. This has a tighter performance guarantee, in the sense that the sequence of X_i it generates is guaranteed to converge if the function satisfies certain conditions.

If the `igLHbest` pivoting mode is being used, then the previously described block of code tested the first X point as the pivot. The next block of code is active if we're in this mode and there are further X points to be tested. It simply repeats the previous block of code for all other pivot choices and keeps track of the best X point generated. Note that this can be expensive computationally, but does not involve any extra applications of the user's function. The aim is to reduce the number of iterations of the imitation game algorithm (via faster convergence to a within-tolerance fixed point), and thus reduce the number of applications of the user's function.

At this point in the execution of the `igImitate()` function's main loop we have generated the next X point to use. We apply the user's function, generate the norm, and append the tuple to the `igList`. Finally, we break out of the loop if we have a within-tolerance fixed point or continue with the next iteration of the main loop.

The main loop may be exited early or its body may be executed `cnt` times. The value of `stat` indicates the reason for the loop's termination. After freeing any unneeded memory, `stat` is used to set the function's termination status, and the function returns.

If the `debug` argument is non-zero (which it is not in all of the supplied example programs), then its bits control the printing of various data internal to the `igImitate()` function. If the least significant bit is set (i.e., the value is odd), then the second player's

strategy probability vectors used to build the X points are printed. These can be used to track whether or not the imitation game algorithm is simply iterating the function or is 'mixing' two or more Y points to generate the X points.

If the second least significant bit is set (i.e., the value is 2 or 3, mod 4), then the `igLHbest` pivoting mode (if it's active) is traced. The probabilities and norms for each pivot choice are printed. Note that these will be different only if different initial pivot choices yield different Nash equilibria.

The functions `igGuess()`, `igIterate()`, and `igImitate()` supplied as part of the `IG` package are essentially wrappers which hide the implementation details from the user. They enable a program to look for fixed points to be constructed easily, but their use is not necessary. It is anticipated that users will modify one or more of these finders to meet their needs, or will write their own. The `example1` and `example2` programs described in Section 6 illustrate how the finders can be used directly to build application programs.

4.6 `iggsl.h`

In a small enough neighborhood of a fixed point, function iteration and the imitation game algorithm typically give a geometric rate of convergence. Newton's method, on the other hand, doubles the number of significant digits on each iteration if the function is sufficiently smooth. Thus the first two algorithms can work well as ways to get a rough estimate of the fixed point, after which Newton's method can be used to rapidly improve the accuracy. The function

```
igFPSTAT igPolish(igList *igl, int cnt);
```

computes a sequence

$$(X_1, Y_1, \|X_1 - Y_1\|), \ldots, (X_{i^*}, Y_{i^*}, \|X_{i^*} - Y_{i^*}\|)$$

where X_1 is the X-value of the last triple in `*igl` (the function fails if the length of `*igl` is zero) for each $i = 2, \ldots, i^*$ the point X_i is the result of applying Newton's method at X_{i-1}, and each Y_i is the value of the function at X_i. Here i^* is either the first i such that $\|X_i - Y_i\| <$ `igTol`, in which case $(X_{i^*}, Y_{i^*}, \|X_{i^*} - Y_{i^*}\|)$ is appended to `*igl`, or `cnt`.

The Nitty Gritty: The `igPolish()` function is a simple wrapper, in the style of `igGuess()`, `igIterate()` and `igImitate()`, that allows approximate fixed points to be improved using the root-finding routines of GSL (Section 35 of the *GSL Manual*). The `example1a` and `example2a` programs described in Section 6 illustrate how the `igPolish()` routine can be used directly by application programs.

The `gslFunc()` in `iggsl.c` is private to that file, and wraps up the user's function (i.e., `igFunction()`) into the form required by GSL. To turn a fixed point problem into a root-finding problem, we simply subtract the X point from the value returned by the user's function (cf., the `example1` program, Section 6.1). The return value from the user's function is translated into its GSL equivalent, while the structure parameter `*p` provided by GSL is ignored. Note that the underlying data in GSL's `gsl_vector` type is simply an array of doubles, and can be accessed directly.

The function `igPolish` takes a non-empty `igList` and iterates the GSL multidimensional root-finding routine (see the *GSL Manual*) starting at the last X point on the list, in an attempt to improve an approximate fixed point to a within-tolerance fixed point. The `cnt` argument specifies the maximum number of iterations allowed. The returned status value indicates whether a problem means that the list was not updated (status `igFail`), or that the list was updated with a fixed point (status `igFP`) or an updated approximation (status `igOK`).

After setting up the GSL routines and data structures, the `igPolish()` function simply iterates the root-finding routine the requisite number of times. The result of each iteration is checked for iteration or numerical problems; these cause either an immediate failure return or a break out of the iteration loop. If a within-tolerance fixed point has been found, the `igList` is updated and an immediate success return is made.

If the iteration loop terminates or is aborted, then the status of the `gsl_multiroot_fsolver` is queried to see if an approximate or within-tolerance fixed point is available. The code here is similar to that in the iteration loop; note the use of the `FREEIT` code block to reduce source code clutter.

The number of calls to the user's function by the `igPolish()` function cannot be accurately predicted. The GSL routines use variants of Newton's method, and the Jacobian has to be approximated using finite difference methods. (The derivative of the user's function is generally not available.) Furthermore, iterations of the root-finding routine may be used to adjust internal parameters or to recompute the Jacobian instead of computing a new approximation. Typically there is: an initial function application; one application for each dimension, for the initial Jacobian; and two applications per iteration loop, one for the root-finder and one to check the result. This yields a rough lower bound, but the actual total number of calls to the user's function may be much higher, particularly for high dimensional problems.

4.7 `igig.h`

The master function used to search for a fixed point is specified here. It has a rather long list of additional parameters, which all have global scope:

```
extern int igigAttempts;     /* number of attempts to find FP */
extern int igigGuesses;      /* number of initial X guesses */
extern double igigDmin;      /* the domain is [Dmin,Dmax]^igDim */
extern double igigDmax;
extern int igigLimit1;       /* max IG size allowed */
extern int igigLimit2;       /* max number of GSL polishes */
extern int igigLimit3;       /* number of IG/polish loops */
```

The function

```
void igigSetup(void);
```

generates and saves a seed for the random number generator, and it specifies the following default values of relevant global variables:

```
igTol = 1.0E-6;          /* epsilon for FP tolerance */
igZero = 4.0E-15;        /* floating-point zero tolerance */
igPLimit = 64;           /* max pivots per Lemke-Howson call */
igLHPivot = igLHlast;    /* use last X point as initial pivot */
igNorm = *igLinfNorm;    /* use L-infinity norm for FP tolerance */
igigAttempts = 10;       /* try to find an FP 10 times */
igigGuesses = 1000;      /* pick initial X's from 1000 guesses */
igigLimit1 = 32;         /* IG sizes up to 32x32 allowed */
igigLimit2 = 64;         /* up to 64 GSL polishing iterations */
igigLimit3 = 16;         /* up to 16 IG/polish loops */
```

Of course you can write code that first runs this function, then resets one or more values that are not to your taste. Note that the `igDim`, `igFunction`, `igigDmin`, and `igigDmax` parameters do not have defaults, and must always be set explicitly.

The master function is

```
igFPSTAT igigFindFP(igList *igl, unsigned int debug);
```

The list `*igl` must be initialised, but may or may not be empty. Any existing data it contains is unaltered. If we find a within-tolerance fixed point we append it to `*igl` and return `igFP`. If there are any problems the list is unchanged and we return `igFail`. If neither of these happen we append the best $\big(X, Y, \|X - Y\|\big)$ triple found to the list and return `igOK`.

The body of the `igigFindFP()` function consists of two nested loops. The outer one makes `igigAttempts` independent attempts to find a fixed point. Each attempt consists of an initial starting guess phase, using the `igigGuesses`, `igigDmin`, and `igigDmax` parameters, and then an inner loop. The inner loop has `igigLimit3` iterations, each of which runs `igImitate()` (with `igigLimit1` as the game size parameter) and then `igPolish()` (with `igigLimit2` as the iteration parameter).

The inner loop starts with the current attempt's guess, and keeps track of the best approximation found in the loop (in the working list) and overall. The inner loop is aborted and the next attempt started if no progress is being made (the `ninit` variable) or the `igImitate()` or `igPolish()` call fails. The working list is cleared for each attempt, and if the outer loop terminates without a fixed point being found then the current best approximation (if any) is appended to the input `igList`. If `debug` is true (i.e., nonzero), then various progress information is printed (see, e.g., Figure 9).

The `demo1`, `demo2` and `example3` programs described in Section 6 illustrate how the `igigFindFP()` wrapper can be used in application programs.

The Nitty Gritty: The `igigFindFP()` function uses its own `igList`, `wList`, as its working list, and does not use the input `igList`, `igl`, except to update it when returning. It also maintains the best approximate fixed point found so far in the `x`, `y` and `n` variables. The `UPDATE` and `MUNGE` code blocks are conveniences to reduce source code clutter. They, respectively, update the best approximation so far from the last entry on the working list, and manipulate the working list so that its sole entry is the best entry.

5 For Advanced Users

At it's simplest, a C function has to be written to implement the user's function and then the top-level `igigFindFP()` wrapper invoked, as described in Section 2.2. If a more sophisticated program is required, or if more control is required over how the features of the `IG` package are combined to find a fixed point, then one of the example programs described in Section 6 can be used as a model.

The next two subsections briefly discuss some points that needed to be considered when using floating-point arithmetic and writing the user function. The final subsection discusses the GSL and some of its features which might be useful.

5.1 Floating-point Arithmetic

The `IG` package is intended for general use and, to ensure ease of use and speed, all calculations are done using floating-point arithmetic (i.e., C's `double` type). The inexact nature of such calculations means that the package has to be coded carefully to avoid system crashes, program non-termination, or the generation of erroneous results. Of course, a putative within-tolerance fixed point is easily verified, so our main concern is that all potential problems are detected and that the routines that contain them 'fail-safely'; i.e., they return normally but with an appropriate failure status.

The core algorithms of the `IG` package are the imitation game algorithm in `igImitate()` and the Lemke-Howson algorithm in `igLemHow()`. The formal statement and proof of these algorithms assume that exact arithmetic is being used; this would, typically, be rational arithmetic. When floating-point arithmetic is used these algorithms can loop indefinitely, generate `INF` (infinite) or `NAN` (not-a-number) floating-point values, become internally inconsistent, or generate incorrect results.

To address these problems, all the core program loops in the package have limits on the number of times they are executed and/or their termination status is checked. Additionally, the coordinates of the X and Y points, applications of the user's function and calculations of the norms are checked to ensure that all the $\langle X, Y, \|X - Y\| \rangle$ tuples added to `igList` structures are valid and do not contain `INF` or `NAN` values.

5.2 The User Function

Writing the user-defined function is obviously an important part of using the `IG` package to find fixed points. The user function is treated as a black box by the `IG` package. The function is given a d-vector of `double` values as its input, and generates a d-vector of `double` values and a status value as its output. The function application may return a failure status value or it may generate `INF` or `NAN` values in the output d-vector. These are detected and handled by the package (see the previous subsection), and so do not cause programs to fail. However, they may mean that a fixed point is not found or that finding a fixed point takes more than one attempt.

In many cases, the user function has a bounded domain, or the domain should be limited to ensure that function calls succeed and generate finite values. However, even if

an initial domain for the X points is specified (e.g., via the `region` option, Appendix A), the `IG` package can generate X points outside this domain (e.g., via function iteration or the root polishing routines). Note that the `IG` package itself does not clip the coordinates of any of the X or Y points generated, although application code is free to do so if required.

Often, a straightforward implementation of the user function suffices. For example, the function of `example3` (Section 6.6) accepts any value in $\mathbb{R}^d$ and has a bounded range. Finding fixed points here is easy, even for high-dimensional spaces. The function of `example1` (Section 6.1) also accepts any value in $\mathbb{R}^d$, but the output is unbounded as x increases. However the growth is only polynomial and a good bound on the region containing the roots is available, so finding a root is generally not difficult. Note that using function iteration (i.e., the `method 2` option) is not a good choice for this example.

The `example2` program (Section 6.2) is couched in terms of normalised vectors (i.e., bounded domain and range). However, the `IG` package is not 'aware' of this. So the function is written to accept any vector as input, while outputting only normalised vectors. This, together with the enforced failure of the function if the eigenvalue is zero, ensures that the fixed points (i.e., stochastic eigenvectors with positive eigenvalues) are correctly determined.

The individual functions in the `demo1` program have domains of $[0, 1]$; values outside this interval can yield `INF` or `NAN` values, or 'unwanted' fixed points. To address this problem the individual functions which make up the user function are written to detect arguments outside the unit interval and to return the values of $f(0)$ or $f(1)$. This preserves the continuity of the functions and does not introduce any new fixed points.

A similar technique is used in the `demo2` program, where the individual functions go to infinity exponentially fast for large x coordinates, making the upper fixed point of each pair hard to find. The functions are modified to run parallel to the $y = x$ line outside the region of interest, while preserving continuity.

5.3 Using the GSL

The `IG` package makes no use of GSL apart from the multidimensional root-finding routines used to polish approximate fixed points. This is a deliberate choice, since it allows the core functionality of the package to be used if GSL is not available. However GSL has many features which could be used to replace or extend those in the `IG` package. For example, GSL has its own error handling routines (Section 3 of the *GSL Manual*) and implementation of vectors and matrices (Section 8 of the *GSL Manual*). Some further features are discussed below.

The random number generators supplied with the `IG` package (i.e., the `rand()` system function and the `igNormal()` routine) are intended to support non-deterministic generation of example functions and initial X points. These work well, but they "don't produce high-quality randomness and aren't suitable for work requiring accurate statistics" GSL comes with a suite of high-quality random number generators and support for a range of random number distribution.

GSL has a range of basic mathematical functions which can be used as substitutes if the system library versions are not available, as per Section 4 of the *GSL Manual*. In

particular, the `gsl_finite()` routine can be used in place of the system's `isfinite()` routine. Instead of using the `IG` package's `igZero` value to test two values for approximate equality, the GSL routine `gsl_fcmp()` can be used. Note that this uses a relative accuracy parameter ϵ, and is not suitable for testing whether a value is approximately zero.

The `IG` package transforms the fixed point problem into a root-finding problem and then uses the multidimensional root-finding routines of GSL (Section 35 of the *GSL Manual*) in an attempt to improve an approximate fixed point. There are several algorithms for root-finding, and some experimentation with these might yield better results for a specific user function. Note that if the derivatives of the user function are available then there are algorithms which can use these, instead of approximating them using difference methods. GSL also has a range of multidimensional minimisation routines which may be useful for certain problems.

The `igGuess()` function generates 'random' points in d-space to use as initial X points. GSL can generate quasi-random sequences in arbitrary dimensions (Section 19 of the *GSL Manual*). These sequences progressively cover a d-dimensional space with a set of points which are uniformly distributed. Such a sequence might be useful if, for example, it was required to scan a region in an attempt to find all fixed points within it.

6 Example Programs

All of the example programs supplied with the `IG` package are command line utilities. That is, they accept options on their command line, use the standard input and output streams for their I/O, and return an exit status of zero if there are no errors. If there are errors, a message is send to the standard error stream and the program terminates immediately with a non-zero exit status. User errors (e.g., unrecognised options, invalid parameters) are handled directly by the top-level code (e.g., the `errorExit()` function in `example1.c`) and result in a simple "`ERROR: ...`" message. Errors from the `IG` package or related to the user's function (e.g., memory allocation errors, bad arguments) are handled by the `IG` package's `igAbend()` function and result in an "`ERROR ...`" message which includes the source file and line number where they occurred.

The `example1` and `example2` programs use the stand-alone portions of the `IG` package, and illustrate how the package can be used if the GSL is not available. The other examples use GSL, with the `example1a` and `example2a` programs using the `igPolish()` function and the `demo1`, `demo2` and `example3` programs using the `igigFindFP()` wrapper function. The `demo1` and `demo2` examples take no arguments. In the other examples, the arguments can be used to investigate how the behaviour of the `IG` package varies as the function and the control parameters are varied.

The examples are chosen for illustrative purposes, and there's no suggestion that solving these problems using fixed points is an efficient method of doing so. Note that the dimensions and coefficients of most of the examples are deliberately kept small for pedagogic purposes and to limit the amount of output, and that the `IG` package is capable of solving large problems in much higher dimensions.

Consult Appendix A for more details on the example programs' options and their parameters. Use a program's `-h` option to see its available options and their permissible argument values.

6.1 The `example1` Program

This example program, implemented via the `example1.c` file, demonstrates how zeroes of a function can be found using fixed points. Specifically, we find roots of univariate polynomials. To solve the equation $f(x) = 0$, we recast it as a fixed point problem by considering the function $g(x) = x + f(x)$, where x is a zero of f iff x is a fixed point of g.

Given a degree $n \geqslant 1$ and a coefficient limit $m \geqslant 1$, the `f1Init()` function generates a polynomial of degree n where the $n+1$ coefficients are independently drawn at random from the integers in $[-m, +m]$. Any of the coefficients may be 0, and the `f1Triv()` function is used to detect the case where all the non-constant terms are 0; this case is disallowed. The problem space is one dimensional, so the `f1Apply()` function takes a pointer to the x value and updates the y pointer's referent to the value $x + a_n x^n + \cdots + a_1 x + a_0$.

Invoking the program with no arguments is equivalent to typing "`./example1 --attempt 4 --degree 3 --eps 1e-5 --fmode 4 --lh 0 --limit 150 --method 1 --norm -1 --pivot 50 --seed 0 xmode 1000 --zero 1e-14`". The dimension is always fixed

internally at one. If there is no explicit `region` option, then the initial X domain minimum and maximum are set to $\pm$(`fmode`+2). This region is based on the Rouché theorem, which yield a bound of `fmode` + 1. The `degree` and `fmode` parameters must be in the range 1 to 8.

The program generates a random function as described above, and then makes a number of independent attampts to find a root. Each attempt is run to completion, and may yield a root, an approximate root, or fail. Each attempt uses the `igGuess()` function to generate the initial X point, and then uses either the `igImitate()` or `igIterate()` function to look for a fixed point (i.e., a root).

The program prints the banner, the parameters, the polynomial, and the result of each attempt. If a root is found, the x and $f(x)$ values are printed (as the `X` and `Y` fields). The size of the `igList` is also printed. For `igImitate()` this allows the maximum game size to be calculated. For `igIterate()` the list size will always be 2 (assuming no failure).

If no root is found, the x and $f(x)$ values of the best approximation on the list are printed. The suffix on the `X` and `Y` fields indicates the index of these values in the list. This index is frequently 0, indicating that there was no improvement on the initial X value. This is due to the rapid increase of $|f(x)|$ as $|x|$ increases. Although the initial X value is bounded, all the Y values and subsequent X values need not be. Thus, in this example, we are relying on the `IG` library code to detect `INF` and `NAN` values and to fail-safe (see Sections 5.1 and 5.2).

6.2 The `example2` Program

This example program, implemented via the `example2.c` file is a simple demonstration of a problem that can be solved using fixed points. In what follows, a matrix or vector is positive if all its components are greater than zero, is non-negative if all its components are greater than or equal to zero, and is non-zero if at least one of its components is not zero. `Example2` works with matrices with non-negative entries, and finds positive eigenvectors with positive eigenvalues.

Given a $d \times d$ matrix A and a non-zero d-vector $\mathbf{x}$ (both over $\mathbb{R}$), if $A\mathbf{x} = \lambda\mathbf{x}$ for some scalar λ, then λ is an eigenvalue and $\mathbf{x}$ an eigenvector of A. In this case, if $\lambda \neq 0$, then $A\mathbf{x}/\lambda = \mathbf{x}$. Thus, given A and a non-zero λ, an ability to find fixed points can be used to find eigenvectors. However, a general A may have no, one or up to d distinct real eigenvalues, and finding these requires finding the roots of the degree d characteristic polynomial (cf., Section 6.1).

If $\mathbf{x}$ is an eigenvector of A with eigenvalue λ, then $\mu\mathbf{x}$, $\mu \neq 0$, is also an eigenvector of A with eigenvalue λ. Now suppose that $\mathbf{x}$ is non-negative and non-zero, and let $|\mathbf{x}|$ be the sum of $\mathbf{x}$'s components (i.e., $\|\mathbf{x}\|_1$). Then $\mathbf{x}/|\mathbf{x}|$ is also an eigenvector of A, and its components are non-negative and sum to one. This normalised eigenvector is called a stochastic eigenvector.

By the Perron-Frobenius Theorem (see, for example, [10]), if A is positive, then it has a unique stochastic eigenvector. Also, if A is non-negative, then it has at least one non-negative eigenvector. Note that, in the latter case, the associated eigenvalue may be zero.

Now consider the mapping $f : \mathbf{x} \mapsto A\mathbf{x}/|A\mathbf{x}|$, where A and $\mathbf{x}$ are non-negative and $\mathbf{x}$ is non-zero. If $|A\mathbf{x}| \neq 0$, then f is well-defined and the fixed points are the stochastic eigenvectors of A. Given such an eigenvector, the corresponding eigenvalue is given by $|A\mathbf{x}|$. If $|A\mathbf{x}| = 0$, then f is not defined. However, $\mathbf{x}$ is an eigenvector with eigenvalue zero, since $A\mathbf{x} = \mathbf{0}$, and $\mathbf{x}$ can be normalised to yield a stochastic eigenvector.

Given a dimension $d \geqslant 2$ and a maximum value $m \geqslant 1$, the `f1Init()` function constructs a random example of the mapping f described above. It constructs the array A by choosing its entries uniformly at random from the integers in $[0, m]$. Given a non-negative d-vector $\mathbf{x}$ (the function's `x` argument), the `f1Apply()` function calculates $A\mathbf{x}$ and $|A\mathbf{x}|$. If $|A\mathbf{x}| \geqslant z$, where z is the current zero tolerance, then f is taken to be well-defined, the function's `y` vector is updated, and the function returns success. Otherwise, it returns failure. Note that the updated `y` vector is always normalised, even if the `x` vector was not.

The value of $|A\mathbf{x}|$ is stored internally on each call to `f1Apply()`, in the `lambda` variable (which is local to the `example2.c` file). If the input to the function is a normalised vector and the call succeeds, then output vectors which are fixed points represent stochastic eigenvectors, and `lambda` is the eigenvalue. (Of course, there must be no other calls to `f1Apply()` before the value of `lambda` is accessed.)

Invoking the program with no arguments is equivalent to "`./example2 --attempt 3 --dim 3 --eps 1e-6 --fmode 2 lh 0 --limit 125 --method 1 --norm -1 --pivot 25 --seed 0 --zero 4e-15`". The program generates a random problem instance as described above and then attempts to find stochastic eigenvectors, using either imitation games or function iteration (a `method` argument of 1 or 2 respectively). The initial attempt to find a fixed point uses the barycentre (i.e., $[1/d, \ldots, 1/d]$) as the starting point. Subsequent attempts, if any, use non-deterministically generated normalised points. All attempts are run to completion, and may yield either success or failure.

Note that the `example2` program uses only normalised vectors, so both the domain and range are compact (i.e., closed and bounded). Also, no attempt is made to process function applications which fail; i.e., no attempt is made to detect (stochastic) eigenvectors with eigenvalues of zero.

If `dim` and/or `fmode` are 'small' it is not uncommon for there to be no solution (i.e., an eigenvalue of zero), or for the eigenspace to have dimension >1. The edited transcript in Figure 6 illustrates one such example. A has eigenvalues of 1 (with multiplicity one) and 2 (multiplicity two). For the eigenvalue of 2, the eigenvectors have the form $[a, b, a + 2b]$, so the stochastic eigenvectors are $[\alpha, (1 - 2\alpha)/3, (2 - \alpha)/3]$, where $0 \leqslant \alpha \leqslant 1/2$. In 1000 attempts, the `example2` program did not find the eigenvalue of 1 (where the eigenvector is $[0, 0, 1]$), but it did find a wide range of stochastic eigenvectors for the eigenvalue of 2.

6.3 The `example1a` and `example2a` Programs

The `example1a` and `example2a` programs are modified versions of the `example1` and `example2` programs, and illustrate how the GSL can be used to improve an approximate fixed point. If the `example1` and `example2` programs do not find a within-tolerance fixed point, they simply print the best approximation found and then exit. The `example1a`

Figure 6: A run of the `example2` program

```
$ ../example2 --seed 1302156089 --attempt 1000

Find eigenvectors & eigenvalues, using fixed points & imitation games
IG v1.1 is Copyright (C) 2011, The Gambit Project
This is free software, distributed under the GNU GPL

dim = 3  tol = 1e-06  zero = 4e-15  pivot = 25  lh = last
seed = 1302156089  fmode = 2
method = imitate  attempt = 1000  norm = Linf  limit = 125

A = 2 0 0
    0 2 0
    1 2 1

Barycentric attempt ...
  FP, via imitation, |list| = 18
    norm X-Y = 6.10354356500586e-07
    E-value = 2.00000305176713
    E-vect = 0.200000610353425 0.200000610353425 0.59999877929315

Attempt #314 ...
  FP, via imitation, |list| = 18
    norm X-Y = 8.06024530453087e-07
    E-value = 2.00000479371335
    E-vect = 0.00884959994436592 0.327435197941539 0.663715202114095

Attempt #460 ...
  FP, via imitation, |list| = 18
    norm X-Y = 6.35659214265338e-07
    E-value = 2.00000257895692
    E-vect = 0.478874474430072 0.0140845433655903 0.507040982204338
```

and `example2a` programs however apply the `igPolish()` function to this approximation in an attempt to find a better approximation.

The number of polishing loops is controlled by an new `--iter` option, with a default value of 100. The default values of some of the other parameters have been altered; in particular, the default value of `eps` has been reduced to 10^{-9} to better exercise the polishing routine.

The `example1a` and `example2a` programs also keep track of how many time the user-defined function is called, and print this information as part of their output (as the `callCnt` value).

6.4 The `demo1` Program

The demonstration program `demo1`, briefly discussed in Section 2.2, is based on the example non-trivial involutions on the unit interval given in [13, pp. 3–4]. These are $f_0 = 1-x$, $f_1 = (1-x)/(1+x)$, $f_2 = \sqrt{1-x^2}$ and $f_3 = 1-\sqrt{1-(x-1)^2}$. These all have a unique fixed point (in the interval $[0,1]$). They also map $0 \to 1$ and $1 \to 0$, so it is easy to

extend their domain to $\mathbb{R}$ while preserving continuity and the range of $[0,1]$, and without introducing any additional fixed points.

The source code of Figures 2 and 4 is contained in the `demo1.c` file. The output of the program is simply the contents of the $\langle X, Y, \|X - Y\| \rangle$ tuple added to the list by the `igigFindFP()` function. The default norm is the ℓ_∞-norm, so the differences between the $X[\cdot]$ and $Y[\cdot]$ values are at most the printed `norm` values.

Note that since the functions are involutions they are 'symmetrical' about the $y = x$ line; i.e., the points $(x, f(x))$ and $(f(x), f(f(x)) = x)$ are on opposite sides of the $y = x$ line and the line segment joining them is at right angles to the $y = x$ line. Thus the averages of the $X[\cdot]$ and $Y[\cdot]$ values are better approximations that either of the values. The numerical values of the fixed points, rounded to fifteen decimal places, are: 0.5, 0.414213562373095, 0.707106781186548 and 0.292893218813452.

6.5 The `demo2` Program

This example, implemented by the `demo2.c` file, is based on the discussion in [4] of the exponential function $E_\lambda(z)$. The functions constructed have (at least) 2^d fixed points, and some care in applying a function (i.e., programming the user's function) is required if all of these are to be found reliably.

We start with the 1-dimensional case. Given a $\mu > 0$, set $\lambda = \mu/e^\mu$, and consider the graphs of $y_1 = x$ and $y_2 = \lambda e^x$. Then $y_1 = y_2$ at $x = \mu$, and we have a fixed-point for the function $f(x) = \lambda e^x$. Note that $\mu > \lambda > 0$ and that λ has a maximum value of $1/e$ when $\mu = 1$. If $\mu = 1$ then, since $f'(x) = \lambda e^x$, $f'(\mu) = 1$ and so y_1 and y_2 are tangent at $x = \mu$ and this is the only fixed-point.

Now suppose that $\mu > 1$. Then $f'(\mu) = \mu > 1$. Since $f(\mu) = \mu > f(0) = \lambda > 0$ and $f(x)$ is continuous, the graphs of y_1 and y_2 must intersect at some point in the open interval $(0, \mu)$, and $f(x)$ has another fixed point. In fact, the second fixed point is in the interval $(0, 1)$, since $f(1) = \mu/e^{\mu-1} < 1$.

Thus if we take any $\mu > 1$ and let $\lambda = \mu/e^\mu$, then the function $f(x) = \lambda e^x$ has precisely two fixed-points. One is at $x = \mu$ and the other is in the open interval $(0, 1)$. As examples, if $\mu = 1.01$, then $\lambda = 0.367861169$ and $f(0.990066225) = 0.990066225$, while if $\mu = 10.0$, then $\lambda = 0.000453999$ and $f(0.000454206) = 0.000454206$ (all results rounded to nine decimal places). Note that λ is small for large μ and we have $f(\lambda/(1-\lambda)) \approx \lambda/(1-\lambda)$, since $e^x \approx 1 + x$ for small x.

To extend this to d dimensions consider the vector function $f(X) = Le^X + M(X - Le^X)$, where L and X are d-vectors and M is a $d \times d$ matrix. The Le^X term stands for d independent 1-dimension problems, where $L = [\lambda_1, \ldots, \lambda_d]$ and $e^X = [e^{X[1]}, \ldots, e^{X[d]}]$. The second term is a commingling factor so that $f(X)$ is not simply d independent problems (cf., Section 6.4). This function obviously has at least 2^d fixed-points.

The `f2Init()` function generates a random instance of such a function. It draws the μ_i independently at random from the uniform distribution on the open interval $(1, 4)$, and draws the entries of M independently at random from the normal distribution with mean 0 and standard deviation $1/8$.

Figure 7: A run of the `demo2` program

```
  mu[] = 3.68464046600275  3.07354835071601  3.46921476838179
lamb[] = 0.092507319116201  0.142172290710356  0.108036462665631
 M[][] = -0.289752067261762  0.0267101995717575  -0.0543487025091722
       = -0.0581314091845104  0.0204003224633143  -0.272639637238498
       = 0.0685642426869546  -0.0301070892669694  -0.0134784463553595

Found a fixed point, norm = 3.10862446895044e-15
  X[0] = 0.102491431329441  Y[0] = 0.10249143132944
  X[1] = 3.07354835071601  Y[1] = 3.07354835071601
  X[2] = 0.122062232887129  Y[2] = 0.122062232887131

Number of function calls = 1052
```

The lower-valued fixed point of each pair is easy to find, since it's attractive. However the upper-valued one is more difficult, since it repelling and the function is increasing exponentially. The `f2Apply()` function addresses this issue by 'clipping' the function for larger values of the coordinates of X. If a coordinate, x, is greater than 7, then the corresponding function is modified from λe^x to $\lambda e^7 + (x - 7)$. This runs parallel to and above the $y_1 = x$ line, and preserves continuity without introducing any new fixed points.

The `demo2` program generates a random problem instance of dimension three, sets the domain for `igGuess()` to $[-6, +6]$ and the norm tolerance to 10^{-12}, and then invokes `igigFindFP()` to look for a fixed point. The values of the μ_i and the λ_i and M are printed for reference, along with the fixed point and the norm and the number of function calls (see Figure 7). Note that problems have eight fixed points (at least), and that the one found by the `demo2` program is determined by chance.

6.6 The `example3` Program

This example program, implemented via the `example3.c` file, uses as its exemplar function that used in [9, Example 4.6]. This function is of the form $f(X) = Z - \arctan(M(X-Z)^3)$, where Z is a d-vector and M is a $d \times d$ matrix (both over $\mathbb{R}$). (The original used Y instead of Z, but this conflicts with our usage of Y to represent $f(X)$.) Z and M are determined when the function is generated, and are fixed for each run of the program. The $\arctan(\cdot)$ and $(\cdot)^3$ operations are performed component-wise, and the function obviously has at least one fixed-point, at $X = Z$ (see [9] for more details).

Given a dimension $d \geqslant 1$ and a mode $m \in \{1, 2, 3\}$, the `f3Init()` function constructs the matrix M and the vector Z (called `f3m` and `f3z` within the code). The elements of Z are drawn independently from the standard normal distribution (i.e., mean 0.0 and standard deviation 1.0). The mode parameter sets the generation mode for the matrix M. If $m = 1$, then $M = 3I_d$, where I_d is the $d \times d$ identity matrix. If $m = 2$, then the elements of M are the absolute values of samples drawn independently from the standard normal distribution. If $m = 3$, then the elements of M are chosen independently from a normal distribution with mean 0.0 and standard deviation 1/13.

Invoking the program with no arguments is equivalent to "`./example3 --attempt`

Figure 8: A run of the `example3` program

```
$ ./example3 -d 1

Find a fixed point, via imitation games and root polishing
IG v1.1 is Copyright (C) 2011, The Gambit Project
Incorporating GSL version 1.14
This is free software, distributed under the GNU GPL

dim = 250  fmode = N(0,1/13)  xmode = [-1,1]  debug = 1
seed = 1305700648  lhpivot = last  norm = Linf
limit = 50  pivot = 75  eps = 1e-09  zero = 4e-15
attempt = 5  guess = 250  iter = 50  loop = 8

attempt = 0, best norm = -2 ...
  loop = 0, current norm = 3.55714683974102 ...

Found a fixed point, norm = 3.73479025483903e-12
  fn calls = 260
```

`5 --debug 0 --dim 250 --eps 1e-9 --fmode 3 --guess 250 --iter 50 --lh 0 --limit 50 --loop 8 --norm -1 --pivot 75 --seed 0 --xmode 1.0 --zero 4e-15`". There is no bound on the value of `dim`, but note that for larger dimensions the GSL root-finding routine (aka, the approximate fixed point 'polisher') starts to dominate the running time.

The program generates a random function as described above, and then invokes the `igigFindFP()` function in an attempt to find a within-tolerance fixed point, stopping as soon as one is found. Note that the values of M and Z, and the X and Y values of any (approximate) fixed point, are not printed due to their size; only the norm and the number of function calls are printed. If `debug` is non-zero then logging information is printed by the `igigFindFP()` function, allowing the progress of its inner and outer loops and the current best norm value to be monitored.

In the original function description in [9] two methods of choosing the initial X point were discussed. Either draw the coordinates $X[i]$ independently from the standard normal distribution, or draw them from the uniform distribution on the interval $[Z[i] - \pi/2, Z[i] + \pi/2]$. The `igigFindFP()` function does not support either of these, and simply uses `igGuess()` to draw the $X[i]$ from the uniform distribution on $[-$`xmode`$, +$`xmode`$]$.

The use of $\arctan(\cdot)$ in the definition of $f(X)$ means that the function is 'well-behaved' and that no special precautions are needed when writing the user function or using the `IG` packge. In fact, the `example3` program solves problems generated using the default parameters easily, typically in the imitation game phase of the first loop of the first attempt and with a norm much smaller than the `eps` value. To see this, use the `-(-d)ebug 1` option, as in Figure 8. However, if the dimension is increased to 500 then the problem, while still easy more often than not (741 out of 1000 test runs), sometimes becomes much more difficult (248 times, see the edited transcript in Figure 9), and sometimes is not solved (11 times).

Figure 9: Another run of the `example3` program

```
$ ../example3 -d 1 --dim 500

attempt = 0, best norm = -2 ...
  loop = 0, current norm = 4.05908329217662 ...
    imitate norm = 1.23948198742454
    polish norm = 0.0562300868515025
  loop = 1, current norm = 0.0562300868515025 ...
    imitate norm = 0.0562300868515025
    polish norm = 0.028621778920471
  loop = 2, current norm = 0.028621778920471 ...
    imitate norm = 0.028621778920471
    polish norm = 0.0251955326270508
  loop = 3, current norm = 0.0251955326270508 ...
    imitate norm = 0.0251955326270508
    polish norm = 0.0251955326270508
    fail (improvement)
attempt = 1, best norm = 0.0251955326270508 ...
  loop = 0, current norm = 4.13363393420205 ...

Found a fixed point, norm = 0
  fn calls = 12025
```

```
no approximations as yet
  best norm from igGuess()
    igImitate() improves norm
    igPolish() improves norm
  best norm from previous loop
    igImitate() does not improve
    igPolish() improves norm
  best norm from previous loop
    igImitate() does not improve
    igPolish() improves norm
  best norm from previous loop
    igImitate() does not improve
    igPolish() does not improve
    this attempt abandoned
best norm from previous attempt(s)
  best norm from igGuess()
    igImitate() produces solution
```

A Example Program Command Line Options

In this appendix we describe the general features of all the command line options for all the example programs supplied as part of the `IG` package (the demonstration programs do not have any such options). The options are listed below, in alphabetical order. Any required parameters are indicated by `A` and `B`. Default values, specific to each program, are used for any options not present on the command line. Options with a short form are indicated by, e.g., `-(-h)elp`. A parenthesised list such as "(1, 1a, 3)" indicates that this option is available in the `example1`, `example1a` and `example3` programs, but not in any of the other example programs. The exact meaning of an option may vary between different programs; consult the appropriate section for the specifics.

`--attempt A` (1, 1a, 2, 2a, 3)
Once the function is set up, up to `A` independent attempts are made to find one or more fixed points of it. The program may terminate as soon as one within the nominated `eps` is found or it may continue with the next attempt.

`-(-d)ebug A` (3)
The argument `A` is treated as an unsigned integer, with each bit controlling the printing of some specific extra information during a run of the program. Only the `example3` program currently implements this feature. The argument is passed to the `igigFindFP()` function, where any non-zero value causes it to output logging information which tracks its progress.

`--degree A` (1, 1a)
Sets the degree of the polynomial to be generated.

`--dim A` (2, 2a, 3)
The X_i and Y_i points representing inputs and outputs of the function are vectors in $\mathbb{R}^d$, for some dimension d. This option sets the value of d to `A`.

`--eps A` (1, 1a, 2, 2a, 3)
The process of finding a fixed point of the function is regarded as having succeeded when $\|X - f(X)\| < \epsilon$, where the norm used depends on the `--norm` option. This option is used to set the value of ϵ. Any positive number can be used, but note that the norm is calculated and checked using the C `double` type, so the usual floating-point caveats apply.

`--fmode A` (1, 1a, 2, 2a, 3)
When the function is generated internally by the program, as opposed to being set by the user, the `A` parameter sets the function mode to be used. This may set the form of the function or the range of values to be used.

`--guess A` (3)
This option sets how many guesses the `igGuess()` function makes for the initial X value. (Normally we'd use the `--xmode` option for this, but that's used for another purpose in `example3.c`.)

`-(-h)elp` (1, 1a, 2, 2a, 3)
If this option is present then a help message is printed and the program exits. The help message lists all the options for the invoking program, with a brief description and an indication of the allowable values.

`--iter A` (1a, 2a, 3)
This option sets the number of iterations in the `igPolish()` function. That is, the limit on the number of times that the GSL root-polishing function will be called.

`--lh A` (1, 1a, 2, 2a, 3)
This option sets the method used in the `igImitate()` function to select the initial pivot to be used when running the Lemke-Howson algorithm on an imitation game. The initial pivot is selected from the list of previous X values, and possible values for `A` are: 0, use the most recent X; 1, use the X with the minimum $\|X - f(X)\|$ norm; and 2, try all initial pivots, and use the X on the list which yields as the next X value that closest to the most recent X.

`--limit A` (1, 1a, 2, 2a, 3)
This option sets an upper limit on how much work the `igIterate()` and `igImitate()` functions do in an attempt to find a fixed point. The `A` parameter sets the size of the largest imitation game build and solved (for `igImitate()`) or the maximum number of function calls (for `igIterate()`).

`--loop A` (3)
This option sets the number of iterations of the inner loop for each attempt to find a fixed point (i.e., the outer loop) in the `igigFindFP()` function. Each iteration calls the `igImitate()` function and then `igPolish()` function.

`--method A` (1, 1a, 2, 2a)
This option selects which of the `igImitate()` and `igIterate()` functions to use. Parameters of 1 and 2 select, respectively, the imitation game method and the function iteration method.

`--norm A` (1, 1a, 2, 2a, 3)
This option sets the norm used to check whether or not we're within the required tolerance of a fixed point (see `--eps`). Parameters of -1, 1 and 2 select, respectively, the ℓ_∞-norm (the maximum norm), the ℓ_1-norm (the taxicab or Manhattan norm) and the ℓ_2-norm (the Euclidean norm). Note that this option does *not* affect the payoff matrices for any imitation games, which always use the ℓ_2-norm.

`--pivot A` (1, 1a, 2, 2a, 3)
The Lemke-Howson algorithm, used to find a Nash equilibrium of an imitation game, typically needs only a few pivot operations before a solution is found (see, e.g., [2]). However, in some cases an exponential (in the game size) number of pivots is required, resulting in a much increased running time. Furthermore, as discussed for the `--zero` option, numerical problems can result in infinite pivoting. To address both these issues, all calls to the Lemke-Howson algorithm are given a limit (i.e., `A`) to the number of pivot operations allowed. If this is reached the call is terminated, with no solution. Note that `A` must be at least two, since at least two pivot operations are needed to move from one Nash equilibrium (including the artificial one) to another.

`--region A B` (1, 1a)
This sets the 'region of interest' to the interval $[\texttt{A}, \texttt{B}]$ or, more generally, to the d-cube $[\texttt{A}, \texttt{B}]^d$. This region is the domain for the initial X point, via the `igGuess()` function.

`--seed A` (1, 1a, 2, 2a, 3)
In the absence of this option the random number generator is seeded with a value derived

from the date/time, the processor time used, and the process ID number. If the option is present, the seed value `A` (a non-zero unsigned integer) is used instead. The actual seed value used in the `igSeedRand()` call is included as part of a program's output, so it can be used to repeat a run if required.

`--xmode A` (1, 1a, 3)
This option determines how the initial X point is generated. It either sets how many guesses the `igGuess()` function makes, with `A` being a positive integer, or it sets the domain for `igGuess()` to $[-\texttt{A}, +\texttt{A}]$ (cf., `--region`).

`--zero A` (1, 1a, 2, 2a, 3)
The user's function uses the C `double` type (i.e., fixed precision floating-point numbers), and all of the `IG` package's calculations are performed using these. Calculations using the `double` type can cause various numerical problems, as they are not exact. For example, a floating-point implementation of the Lemke-Howson algorithm may loop indefinitely, despite the fact that, formally, the algorithm is guaranteed to terminate.

In an attempt to mitigate (but *not* solve) this and similar problems, a zero tolerance value ζ (set by the parameter `A`) is used to detect numbers or differences that are 'small' (i.e., in the interval $[-\zeta, +\zeta]$) and to treat these as if they were zero. Any positive number can be used for `A`, but note that normalised `double` values typically have only $\approx$16 decimal digits of precision.

The principal use of the `--zero` value is in the Lemke-Howson code for the imitation games. For this code, the payoffs of the games are offset so that the least value is 1.0. Thus values for `A` of $\approx 10^{-15}$ are appropriate.

References

[1] X. Chen and X. Deng. Settling the complexity of two-player Nash equilibrium. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, pages 261–272, 2006.

[2] Bruno Codenotti, Stefano de Rossi, and Marino Pagan. An experimental analysis of the Lemke-Howson algorithm. *CoRR*, abs/0811.3247v1, 2008.

[3] C. Daskalakis, P. Goldberg, and C. Papadimitriou. The complexity of computing a Nash equilibrium. In *Proceedings of the 38th ACM Symposium on the Theory of Computing*, 2006.

[4] Robert L. Devaney. Cantor and Sierpinski, Julia and Fatou: Complex topology meets complex dynamics. *Notices of the AMS*, 51:9–15, 2004.

[5] Paul Goldberg, Christos Papadimitriou, and Rahul Savani. The complexity of the homotopy method, equilibrium selection, and Lemke-Howson solutions. In *Proceedings of the 52nd Annual IEEE Symposium on the Foundations of Computer Science*, 2011.

[6] M. Hirsch, C.H. Papadimitriou, and S. Vavasis. Exponential lower bounds for finding Brouwer fixed points. *Journal of Complexity*, 5:379–416, 1989.

[7] R. D. McKelvey and A. McLennan. The computation of equilibria in finite games. In *Handbook of Computational Economics*. Elsevier, New York, 1996.

[8] R. D. McKelvey, Andrew McLennan, and Theodore Turocy. Gambit: Software tools for game theory. `http://www.gambit-project.org`, 2010. version 0.2010.09.01.

[9] A. McLennan and R. Tourky. From imitation games to Kakutani. Mimeo, University of Minnesota, 2005.

[10] C. D. Meyer. *Matrix Analysis and Applied Linear Algebra*. SIAM, New York, 2001.

[11] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, Cambridge, 2nd edition, 1992.

[12] R. Savani and B. von Stengel. Hard-to-solve bimatrix games. *Econometrica*, 74:397–429, 2006.

[13] Yu. A. Shashkin. *Fixed Point Theorems*. American Mathematical Society, Providence, 1991.