# `rgsolve` for Java

Richard M. Katzwer
Dept. of Economics
Princeton University
08/04/13

## 1 Introduction

This is documentation for the `rgsolve` package for Java. The package consists of a set of tools for solving two-player repeated games with perfect monitoring and public randomization. These tools are primarily an implementation of the algorithm described in Abreu & Sannikov (2013), hereafter "AS". The algorithm of AS it itself a refinement of of the methods developed in Abreu, Pearce & Stachetti (1990), hereafter "APS". This program was designed by Dilip Abreu, Yuliy Sannikov, Benjamin Brooks, Richard Katzwer and Rohit Lamba. The program was implemented in Java by Richard Katzwer

Please direct inquiries for further information, comments, or bug reports to:

Dilip Abreu, dabreu@princeton.edu
Benjamin Brooks, babrooks@princeton.edu
Richard Katzwer, rkatzwer@princeton.edu
Yuliy Sannikov, sannikov@princeton.edu

## Contents

# 2 Quick-Start Guide

This section is intended to help the user jump right in to loading and solving games with `rgsolve`.[1]

## 2.1 Installation and Execution

You can download the most current version of `rgsolve` at the Princeton Economic Theory Center website, at

http://www.princeton.edu/econtheorycenter/research-links/dynamic-games-algorithm/

The website will direct you to download the file `rgsolve.zip`. The entire program and its resources is contained in this runnable `.jar` file contained in this `.zip` file. [2] The unzipped directory should have the following contents:

- `rgolsve.jar`, the Java executable of the repeated games software.
- A directory marked `doc` which contains the documentation ("Javadocs") on the package's classes and methods.
- This pdf, named `rgsolveUserGuide.pdf`.
- A MATLAB scrip `JavaRGSolve.m` that acts a wrapper for calling the `rgsolve` Java package from within MATLAB.

To run `rgsolve`, your machine needs to have a Java 6 (or higher) runtime environment installed. This can be downloaded at

http://www.oracle.com/technetwork/java/javase/overview/index-jsp-136246.html

and can be installed on almost all systems (Windows, Mac, Linux, etc.).

`rgsolve` can be run[3] by double-clicking the file `rgsolve.jar`. When you open `rgsolve`, you will be greeted with the welcome screen. Click through this screen to continue to the application.

Hover your mouse over components of the program; for most components, a tool-tip will pop up explaining the function of the component.

## 2.2 Example: Solving the Repeated Prisoner's Dilemma

### 2.2.1 Entering the Game

You will see the main window of `rgsolve` as in figure 2.

To enter the Prisoner's Dilemma using the user interface, make sure you are on the Game tab.

1. Set the dimensions of the game to $2 \times 2$ by entering the value 2 in the fields marked No. of Player 1 Actions and No. of Player 2 Actions. This will re-size the table in the Stage Payoffs tab to a $2 \times 2$ game.

[1] For detailed discussion of how to use the GUI and the full flexibility of `rgsolve`'s capability for loading and saving games, see sections 3 and 4, respectively.

[2] Additionally, I will be keeping some resources related to `rgsolve` (such as the Javadocs, and a web-applet) on my website, at the directory http://www.princeton.edu/~rkatzwer/rgsolve

[3] The first time `rgsolve` is run, it will unpack some directories and resources into whatever directory the `rgsolve.jar` file was placed in.


Figure 1: Splash screen

2. Set the discount rate for the players. In this example, we will use $\delta = .7$. In the field marked $\delta$ [Discount], enter the value 0.7.

3. Next, enter the game payoffs. The table stores payoffs in the form $< g_1, g_2 >$ where $g_i$ is the payoff to player $i$.[4] Payoffs should be separated by a comma. For this game, enter the matrix

| | 1 | 2 |
|---|---|---|
| 1 | 2, 2 | -1, 3 |
| 2 | 3, -1 | 1, 1 |

Figure 3: Prisoner's Dilemma Payoffs

4. We are now ready to solve the Prisoner's Dilemma. Press the button marked Solve to start the solver.

### 2.2.2 Exploring the Solution

Your game should now be solved! The first thing to notice is that the top panel has changed, and should look something like in figure 5.

- Time (s) displays how many seconds the algorithm took to converge. In our example, it took .004 seconds.
- Iterations is the number of iterations to achieve convergence, in this case 1 iteration.
- Error is the maximum distance between vertices between the last two iterates before the algorithm stopped. In this example, the set converged exactly.
- Gen. Pts. displays the number of potential extreme points of $V^*$ examined over the course of the algorithm, in this case 12.

[4] Player 1 is always taken to be the row player, and player 2 is taken to be the column player.

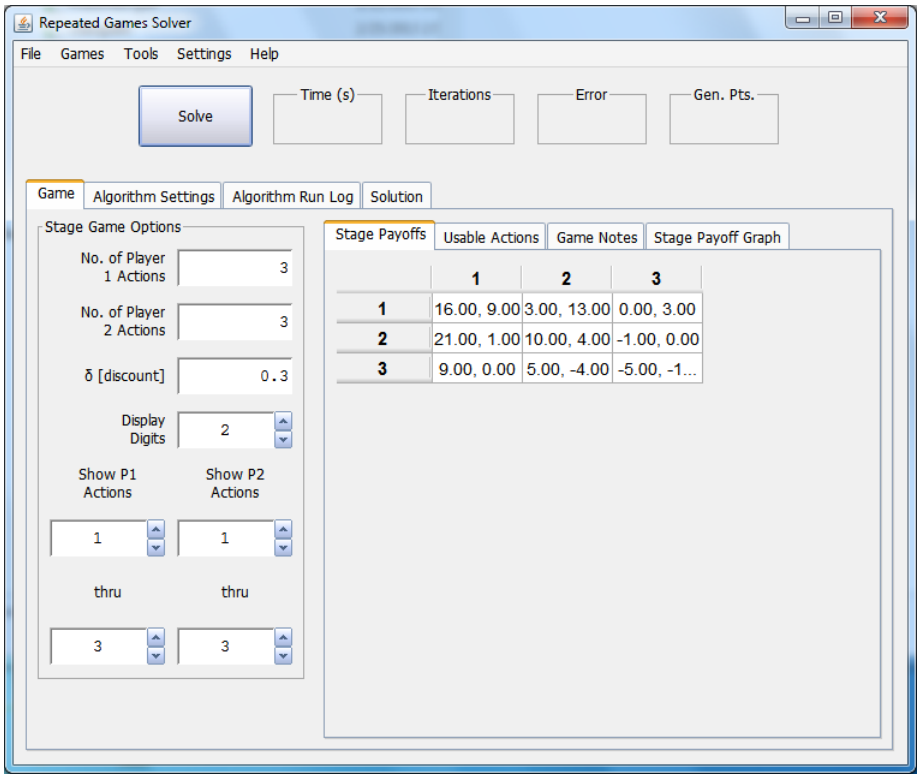


Figure 2: Main `rgsolve` window


Figure 4: Solve Button

The program will now automatically change panels so that you see the Solution tab instead of the Game tab. This panel contains the solution to our Prisoner's Dilemma (see figure 6).

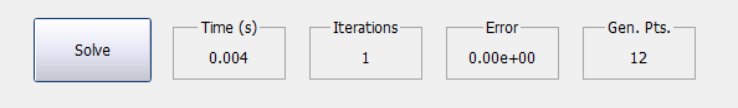


Figure 5: Basic Solution Information

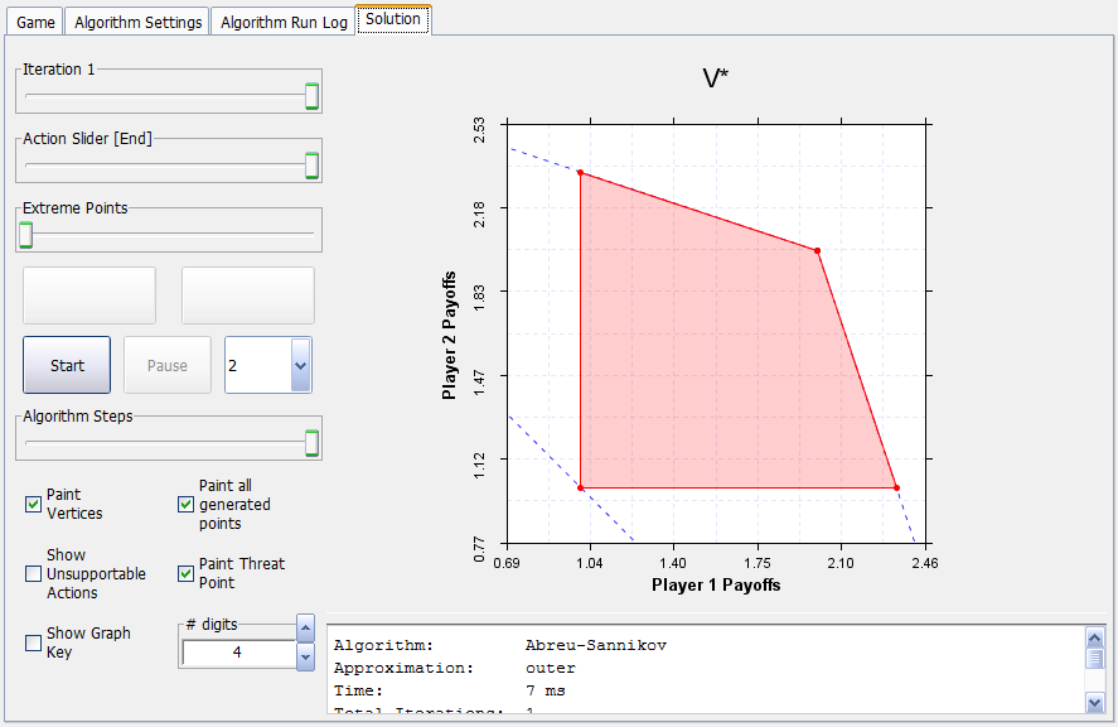


Figure 6: The Solution to our Prisoner's Dilemma

This panel is divided into three components:

- A *control panel* on the left with checkboxes and sliders for exploring the solution.
  - Use the sliders to change the displayed iteration or action profile.
  - Press Start to watch the operations of the entire algorithm from start to finish.
- A *graph* on the upper-right displaying the equilibrium set visually.
  - Zoom in by pressing and dragging your mouse from the upper-left to lower-right.
  - Zoom out by double-clicking the graph.
  - Click on any vertex of the equilibrium set to show information and display graphically which action supports it, what continuation values it involves, etc.
- A *text area* on the lower-right displaying information about what is currently displayed on the graph.

For example, clicking on the right-most vertex of $V^*$ [the point $(2.333, 1)$] changes the graph to display the decomposition of this extremal equilibrium as in figure 7.

The bright green circle on the lower right is the game payoff from the action $(2, 1)$ [i.e. (Defect, Cooperate)]. The transparent blue square displays the expected continuation value in equilibrium when playing this

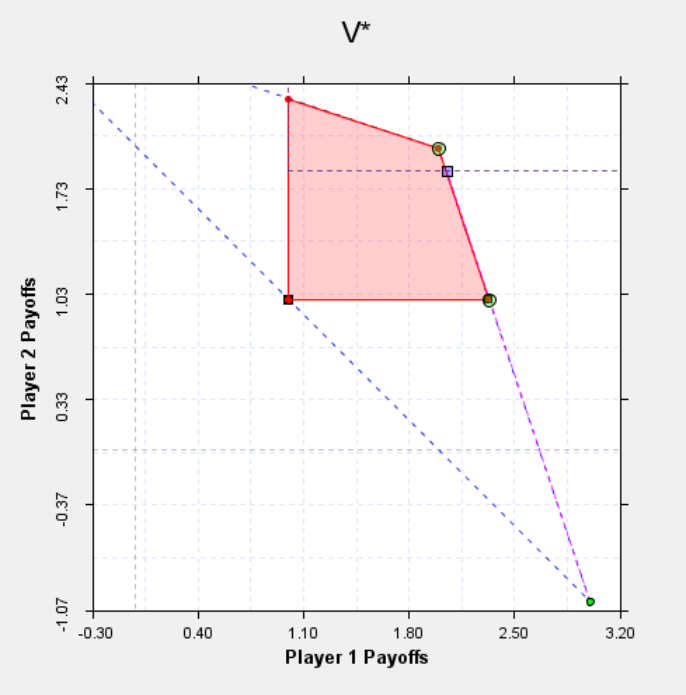


Figure 7: A vertex of $V^*$

action today. The transparent green circles represent the extremal equilibria the players randomize between to achieve the continuation value. The printout in the text area makes this precise. It tells us that with 14.3% probability, we return to the same equilibrium tomorrow (the point $(1, 2.333)$) and with 85.7% probability we go to the extremal equilibrium point $(2, 2)$ which corresponds to playing (Cooperate, Cooperate) forever. The dashed blue L-shape corresponds to the incentive compatibility constraints on continuation values to get players to play (Defect, Cooperate) today: player 1 needs at least 1 tomorrow and and player 2 needs at least 1.857.

`rgsolve` is useful for converting approximate numerical solutions to exact solutions. On the menu bar, select Tools→Mathematica exact solution command. This will bring up a window containing the text

```
s = Solve[
   {  e12 == 3/10 * (-1) + 7/10 * (3/7 * 2 + e12),
      e11 == 3/10 * (3) + 7/10 * (alpha1 * (e11) + (1-alpha1) * (e21)),
      alpha1 * (e12) + (1-alpha1) * (e22) == 3/7 * 2 + e12,
      e21 == 2, e22 == 2, e31 == 3/10 * (-1) + 7/10 * (3/7 * 2 + e31),
      e32 == 3/10 * (3) + 7/10 * (alpha3 * (e22) + (1-alpha3) * (e32)),
      alpha3 * (e21) + (1-alpha3) * (e31) == 3/7 * 2 + e31, e41 == 1, e42 == 1},
   {  e11, e12, alpha1, e21, e22, e31, e32, alpha3, e41, e42 } ]
```

which is the command to solve the system of equations defining the *exact* equilibrium set $V^*$ in Mathematica.

## 2.3 Example: Loading a game via rgsolve's I/O routines

One can enter games manually via the table on the Game tab, but this is time consuming for large games. It is faster to define games elsewhere and load them into `rgsolve`. `rgsolve`'s full I/O functionality is described in section 4, but we describe here how to load a simple Bertrand game from a text file. On the menu bar, select File→Open Game, and the dialog in figure 8 should appear.

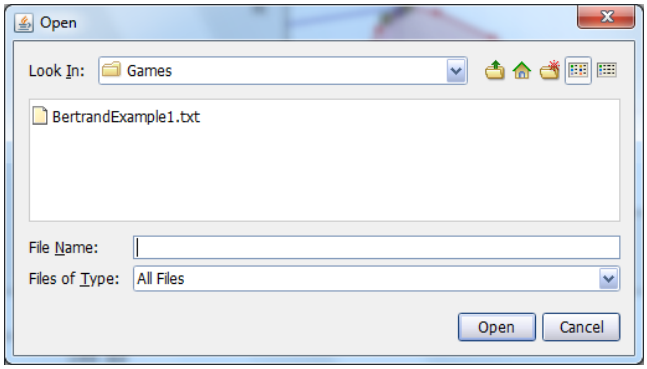


Figure 8: Opening a saved game

We have packaged `rgsolve` with a text file `BertrandExample1.txt` that contains a $20 \times 20$ action Bertrand duopoly game.[5] Highlight the file and press Open. This will load the game from the text file into `rgsolve`, and the game can then be solved like any other. By default, all games are opened from and saved to the `/Games` directory.

## 2.4 `rgsolve` settings

`rgsolve` allows the user to specify settings for the solver. We discuss these settings in detail in section 3.2, but give a brief overview here. On the Algorithm Settings tab, you should see on the left panel the fields in figure 9.

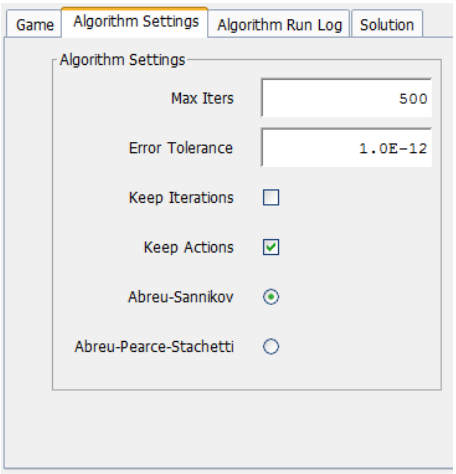


Figure 9: Basic `rgsolve` settings

- `rgsolve` uses iterative methods, that are controlled by the parameters Max Iters and Error Tolerance. Max Iters sets the maximum amount of iterations `rgsolve` should use; the number of iterations necessary can vary greatly in game size and discount factor. Error Tolerance is the convergence criterion of the solver.
- The options Keep Iterations and Keep Actions specify what information from the run of the algorithm `rgsolve` should store for later examination. If you select Keep Iterations, information from *all* iterations

[5] We discuss how to format text files for `rgsolve` compatibility in section 4.3.1.

is stored in memory. Otherwise, only information from the last iteration is retained. Keep Actions specifies what is stored from *within* each iteration. If it is selected, information on the intersection of continuation incentive-compatibility constraints is stored for all supportable actions; otherwise only the payoff sets at the start and end of the iteration are saved (along with how the extreme points of the end-set are generated). You usage of these options should be dictated by the available memory of your system and the dimensions of the game.

- The options Abreu-Sannikov and Abreu-Pearce-Stachetti tell the solver which algorithm to use. Abreu-Sannikov is generally faster and less memory-intensive, although upon convergence the results are indistinguishable.

# 3 Using the `rgsolve` GUI

This section discusses the components of the `rgsolve` GUI and their function in detail.

The main window (figure 2) of `rgsolve` has a panel stretching across the top that contains a button marked Solve and the fields Time (s), Iterations, Error and Gen. Pts. The Solve button activates the solver and computes the solution to the currently displayed game. The fields display information about the solution such as the number of iterations to convergence, etc.

Above this panel is a standard menu bar. In section 3.3, we discuss the menu bar options.

The main window has four tabs:

- Game - Defines the stage game and discount factor.
- Algorithm Settings - Defines solver settings.
- Algorithm Run Log - Prints a log of what the solver is doing as it progresses.
- Solution - Contains an interactive panel for exploring the properties of game solutions

We will discuss Game and Algorithm Settings here, and Algorithm Run Log and Solution later.

## 3.1 The Game Tab

The Game tab is intuitive and contains all the information about the repeated game to be solved.

The left panel (marked Stage Game Options) defines some broad parameters for the game to be solved: the number of actions available to each player and the discount rate $\delta$. These fields are editable, and are used by the user to define the broad bounds of the game. The spinner marked Display Digits indicates the precision of the data displayed in the tab (but not the precision of the underlying data).

The Game tab has four sub-tabs used for exploring the game. They are fairly intuitive, and are

- Stage Payoffs - this tab displays the payoffs of the stage game in the traditional bimatrix format. Using the spinners on the left-panel, you can change which game payoffs are displayed in the table. To keep memory costs low, you can at most view 20 actions at a time.

  The table component on this tab is editable, and you can change payoffs in the game by double-clicking a cell in the spreadsheet and entering a payoff in the form $<g_1, g_2>$.

- Usable Actions - Like Stage Payoffs, this tab contains a table showing which action profiles are "allowed" in equilibrium. That is, if an entry for an action profile is changed from `true` to `false`, the program does not let the players use that profile in equilibrium. By default, games have all action profiles usable (`true`).

- Game Notes - This tab contains a text-field where the user can enter any relevant notes on or description of the current game.

- Payoff Graph - This tab (when selected) generates a graph of the convex hull of feasible stage payoffs in the game. For very large games this may take some time.

## 3.2 The Algorithm Settings Tab

The Algorithm Settings tab in figure 10 controls the execution of the algorithms which solve the repeated game.

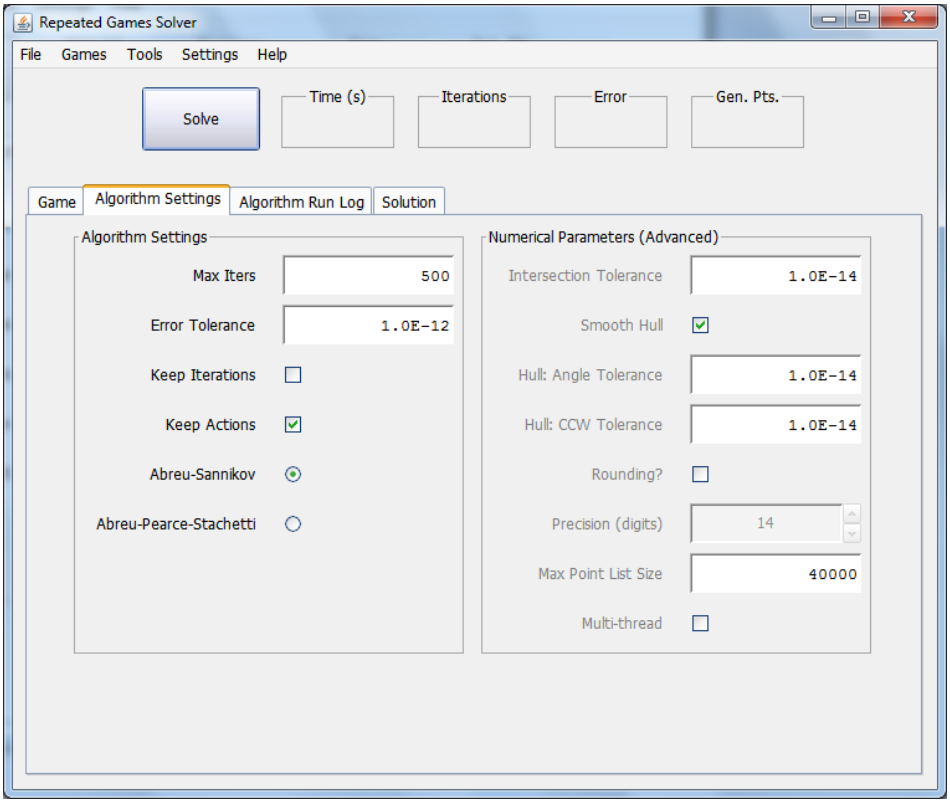


Figure 10: The Algorithm Settings Tab

Parameters are divided into two categories, Algorithm Settings which control some basic settings of the program, and Numerical Parameters, which are more advanced settings relating to some numerical complications encountered in numerical computing of this sort. [Most users should never need to look at the Numerical Parameters panel at all.]

We define each of these settings below:

### 3.2.1 Algorithm Settings

- Max Iters - The maximum number of iterations the solver should do before prompting the user about whether to terminate the algorithm or to use more iterations. (Default: `500`)
- Error Tolerance - The convergence criterion that should be used by the solver. The solver stops (and considers the routine "converged") if (a) the number of extreme points between successive iterates is the same and (b) the (sup-norm) distance between corresponding extreme points in successive iterates is less than this value. (Default: `1e(-12)`)
- Keep Iterations - A switch for whether the program should keep a record of all iterations performed, or just keep what happened in the last iteration. (Default: `false`)
- Keep Actions - A switch for whether or not the program should keep a record of the I.C./Feasible-Set intersections (and thereby a record of all candidate extreme points generated), or just keep the extreme points that survive the convex hull operation. (Default: `true`)
- Abreu-Sannikov and Abreu-Pearce-Stachetti - These are linked radio buttons for selecting which algorithm the solver should use, the methods of Abreu & Sannikov or the methods of Abreu, Pearce and

Stachetti. (Default: `AS`)

### 3.2.2 Numerical Parameters

- Intersection Tolerance - This tells the program the "slack" to give the solver when calculating I.C.'s. This is for cases where an I.C.-constraint is lies right up against the edge of the feasible set, and numerical error would cause the program to not generate the correct points. (Default: `1e(-15)`)
- Smooth Hull - This is a switch for whether the convex hull routine should "smooth" the hull by eliminating extraneous extreme points via certain robustness checks, or whether the convex hull routine should naively return all potential extreme points based on a potentially inaccurate floating point signed area calculating. (Default: `true`)
  - Hull: Angle Tolerance - If is Smooth Hull selected, this tells the convex hull routine how "flat" angles between 3 adjacent points have to be to eliminate the middle point. If the angle formed by the three points is in excess of $\pi$ minus this value, the middle point is deleted. (Default: `1e(-15)`)
  - Hull: CCW Tolerance - If is Smooth Hull selected, this tells the convex hull routine the minimum (robustly calculated) signed area to allow between 3 adjacent extreme points. If the signed area is less than this value, the middle point is deleted. (Default: `1e(-15)`)
- Rounding? and Precision (Digits) - If Rounding? is selected, all calculations in the solver are rounded to the number of digits in Precision (Digits). (Default: `false` and `15`)
- Max Point List Size - Typically, the solver generates many candidate extreme points over the course of an iteration, and then takes the convex hull at the end to delete extraneous points. However, to conserve memory, once the number of candidate points exceeds this number, the hull is taken immediately and the extraneous points (up to that time) are deleted. (Default: `100000`)
- Multi-thread - On systems with multi-core processors, selecting this tells the solver to break up some of the computationally intensive tasks within an iterations and to send each sub-task to a different processor to speed up computation time. Even if selected, the solver will only multi-thread games where the number of action profiles exceeds five million.[6] (Default: `false`)

## 3.3 The menu bar

The menu bar as the following menus and menu-items:

- File
  - This menu contains options for opening and saving games. We discuss these options in greater detail in section 4.
- Games
  - This menu constructs games from certain pre-defined game classes (using the Predefined games submenu), and allows the user to define new games. We discuss these games in detail in section 4.4.
- Tools

[6] Below this value, the speed-up from parallelization does not merit the computational costs of the overhead of multi-threading.

  - Open Solution in New Window - This pops out the currently displayed solution on the Solution tab into a new window.
  - Save the Current Game Solution - This serializes the currently displayed solution on the Solution tab into a `.rgsoln` file that can be loaded later into `rgsolve`. By default, solutions are saved and loaded from the directory `/Solutions`, which is relative to the location of `rgsolve.jar`.
  - Load the Solution to a Solved Game - This loads `.rgsoln` files, setting the Solution tab to display the loaded solution and the Game tab to display the associated game.
  - Mathematica Exact Solution Command - This prints to a new window the Mathematica command for *exactly* solving the system of equations defining the equilibrium set $V^*$.
  - MATLAB Exact Solution Command - This prints to a new window the MATLAB command for *exactly* solving the system of equations defining the equilibrium set $V^*$.

- Settings
  - Save current settings as user defaults - saves the settings in the Algorithm Settings tab to the file `Settings\UserDefaults.params`. These are the settings that are loaded by `rgsolve` on start-up.
  - Load user defaults - This sets the settings to those contained in the file tab to the file `Settings\UserDefaults.param`
  - Reset settings to RGSolve defaults - this loads the default settings stored in the program.

- Help
  - Help [.pdf] - this item loads this User Guide into your default `.pdf` viewer. (Key-accelerator: CTRL+H)
  - rgsolve for Java website - this item loads this project's homepage at the Princeton Economic Theory Center (Key accelerator: CTRL+W)
  - About - About `rgsolve`!

## 3.4 The Solution Tab

Using this tab, the user can explore the properties of the solutions to dynamic games interactively in great detail. The solution panel is shown in figure 11. The best way to learn how to use this panel is to play with it, but we will briefly discuss its functionality here.

On the left side of the panel you have some tools. On the right side is a graph which displays graphical information about the solution. On the lower right is a text area which prints out information about the solution.

The panel to the left of the solution tab has the following tools:

- An iteration slider which allows you to change the iteration that the graph shows (enabled only if the option Keep Iters is selected).
- An action slider which allows you to step through an iteration and see exactly what happens and which potential extreme points are generated at each action profile. If the slider is all the way to the left [right], the graph shows the feasible set at the start [end] of the iteration. (You can only see what happens within the iteration if Keep Actions is selected.)
- When the action slider is all the way to the right, you can use the Extreme Points Slider. Use this slider allows you to examine each extreme point of the feasible set at the end of the iteration. The graph panel will display the action and continuation values that supports the extreme point. The text area will print information about the extreme point.

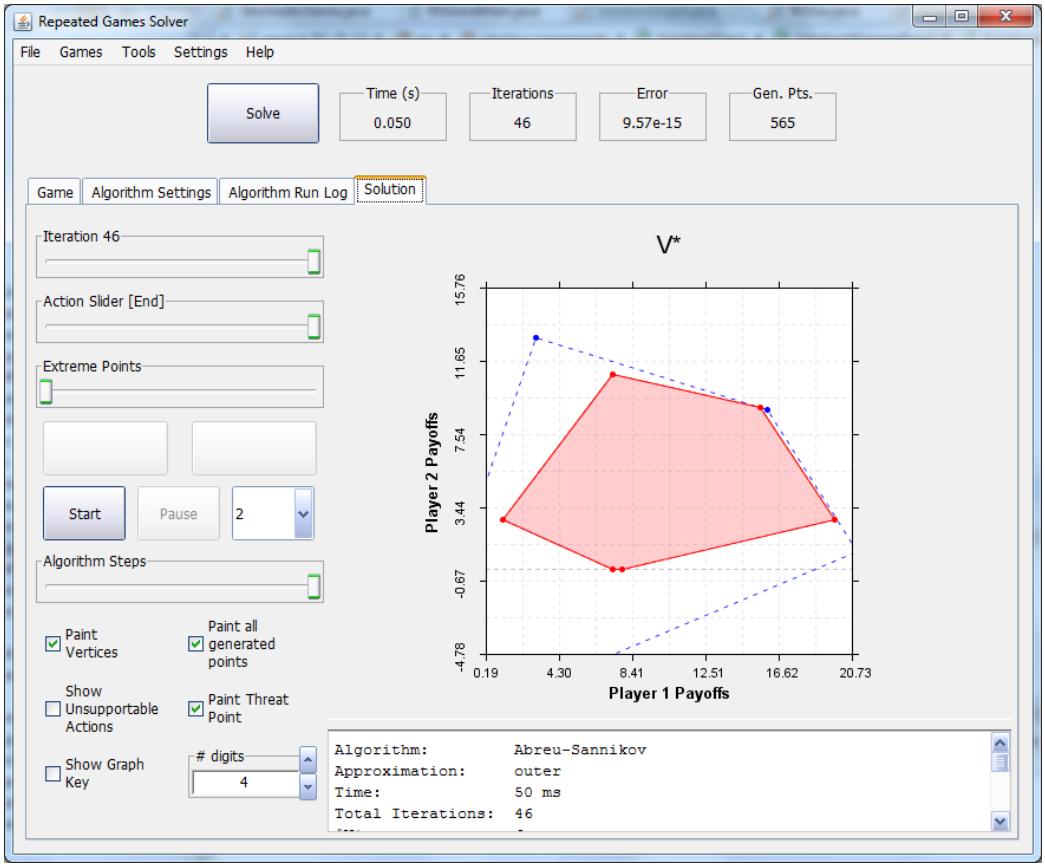


Figure 11: The Solution panel

– Under this slider are two buttons. These buttons are only activated if (i) we are examining $V^*$ on the graph, (ii) an extreme point of $V^*$ has been selected by either clicking on a vertex on the graph or moving the extreme point slider, and (iii) the current extreme point has a continuation that can be decomposed into the randomization between two other extreme points. If these conditions are satisfied, you can use these buttons to "walk through" the equilibrium and see what happens next period.

- Using the Start button, the entire algorithm will be played by back on the graph. You can pause and stop the playback, and set the number of frames per second.
- The checkboxes
  - Paint Vertices decides whether or not the vertices (extreme points) should be painted on the graph.
  - Paint all generated points decides whether all generated potential extreme points in the current iteration should be painted on the graph (up to the current action), or whether the current action should be viewed alone.
  - Show unsupportable actions decides whether the action slider should loop through *all* actions, or just the actions that support equilibrium payoffs.
  - Paint threat point decides whether the threat point in the current iteration should be painted.
  - Show Graph Key toggles the lower-right panel between a graph key which decodes the images on the graph and the text-output defining numerically what you are seeing on the graph.

The graph panel is interactive. You can

- click on extreme points to bring up information on how they are generated.
- zoom in by holding down the mouse and dragging to the lower-right, and zoom out by double clicking.
- By right-clicking you can choose to load the current graph image to the system clipboard (and then paste it anywhere you like).
- By right-clicking you can choose to save the current graph image to the `/Images` directory.

# 4 Writing, Saving and Loading Games

`rgsolve` has very flexible capabilities for saving and loading games. Using the Game tab on the GUI is the quickest way to define small games, but for large games manually inputting each payoff pair is prohibitively time consuming. There are several options here.

By default, Games are saved and loaded from the `/Games` directory, which is relative to the location of `rgsolve.jar`.

## 4.1 The File Menu

### 4.1.1 Opening Games

The File menu has the menu item Open Game (key-accelerator: CTRL+O), which loads a file chooser window. By default, the file chooser opens to the `/Games` directory, which is located relative to the location of the file `rgsolve.jar`. Using the file chooser, you can navigate to and load games from any directory. We discuss which types of files `rgsolve` can load in section 4.3.

### 4.1.2 Saving Games

The File menu has the menu item Save Game (key-accelerator: CTRL+S), which loads a file chooser window. The user then has the By default, the file chooser opens to the `/Games` directory, which is located relative to the location of the file `rgsolve.jar`. Using the file chooser, you can navigate to and save games to any directory. When you choose to save a game, you will be prompted with the dialogs in figure 12, asking which type of file you would like to save the game as. Any game in `rgsolve` can be saved in `.txt` or `.mat` format. Certain games cannot be stores as `.rgm` or `.gmcode` formats. That is why we display the two prompts below; `rgsolve` will only let you save in formats compatible with the loaded game.

In general, `.rgm` format is the most efficient and most compact way of storing games as when possible it saves payoff rules rather than the actual payoff matrices as in `.txt` or `.mat`. The `.gmcode` format is for special games generated through `rgsolve` with user-defined Java rules for payoffs. We discuss this special case in section 4.3.4.

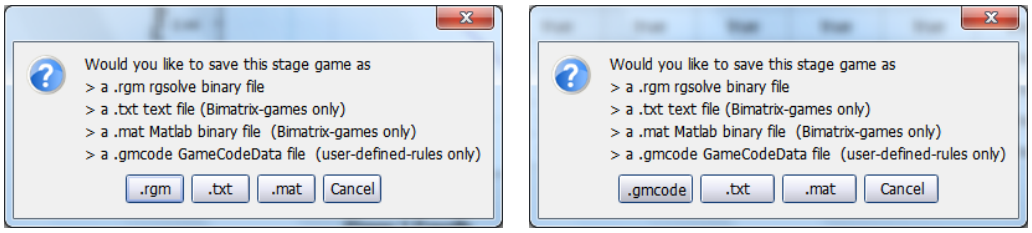


Figure 12: Prompt for file save types

## 4.2 The Games Menu

This menu is for loading/writing new games to `rgsolve`. The submenu Predefined Games allows the user to load the games discussed in section 4.4. The Games menu also has the options enter game payoffs as text matrices and enter game payoff functions as Java code, which we discuss below.

### 4.2.1 Inputting payoff matrices

Selecting the enter game payoffs as text matrices menu item, you can directly copy-paste payoff matrices in text form into `rgsolve`. Selecting this item brings up the window in figure 13. Matrices should have entries delimited by space characters, with each row on a new line. In figure 13, we show how to enter a simple Prisoner's Dilemma. `rgsolve` will conform the inputted matrices to the number of actions set in # Actions 1 and # Actions 2: that is, if the matrices are too small `rgsolve` will pad them with zero-entries, and if they are too large the matrices will be truncated.

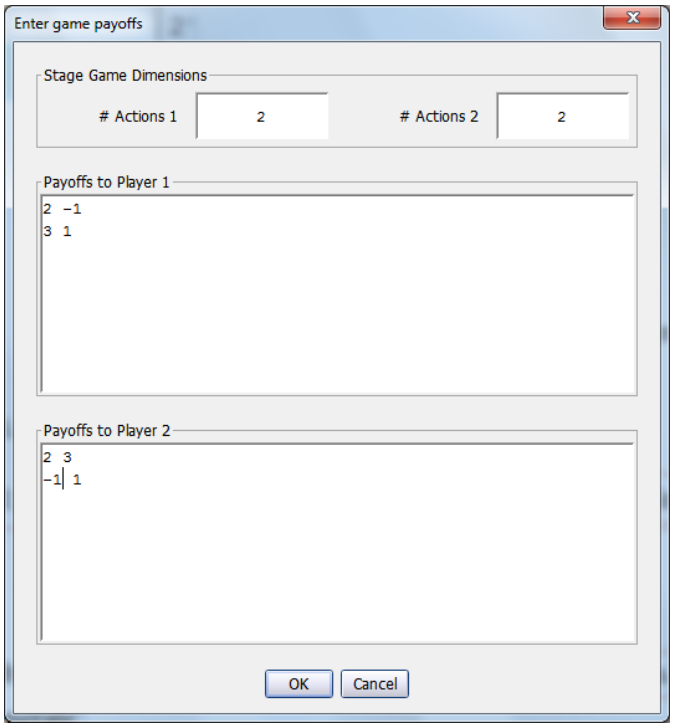


Figure 13: Enter payoff matrices as text

### 4.2.2 Inputting payoff rules as Java code [ ADVANCED! ]

Selecting the enter game payoff functions as Java code menu item brings up the the dialog in figure 14. `rgsolve` is bundled with a Java compiler, which allows it to act as an interpreter and dynamically create and load custom games for users. A user familiar with Java (or C/C++) syntax should use this option to generate interesting games. The fields $m_1, m_2$, and discount are straightforward. In the field Fields, define any "global" variables that you want your payoff functions to use. In the field Constructor, write any statements that you want the constructor to execute (after a call to the `super` constructor of course!). All statements should be on their own lines and terminated by a semicolon (;). In the fields 1's Payoff Function and 2's Payoff Function, define rules for computing game payoffs. You can put in multiple lines here, but

- the payoff functions have the signature

  ```
  @Override public double payoffi(int a1, int a2)
  ```

  so your defined payoffs may depend on the actions, which are the `int` variables $\texttt{a}_1$ and $\texttt{a}_2$. Be aware that although `rgsolve` displays actions $\texttt{a}_\texttt{i}$ as ranging from `1` to $\texttt{m}_\texttt{i}$, within the source of the program actions range from `0` to $\texttt{m}_\texttt{i} - \texttt{1}$.

- make sure your last line involves returning a `double`.
- only use variables that you declared in Fields, or locally (within the scope of the method) defined variables.
- to reference primitives of the Game super-class, use `m1()`, `m2()` and `delta()` to access the number of actions available to player 1 and 2 respectively, and the discount rate.

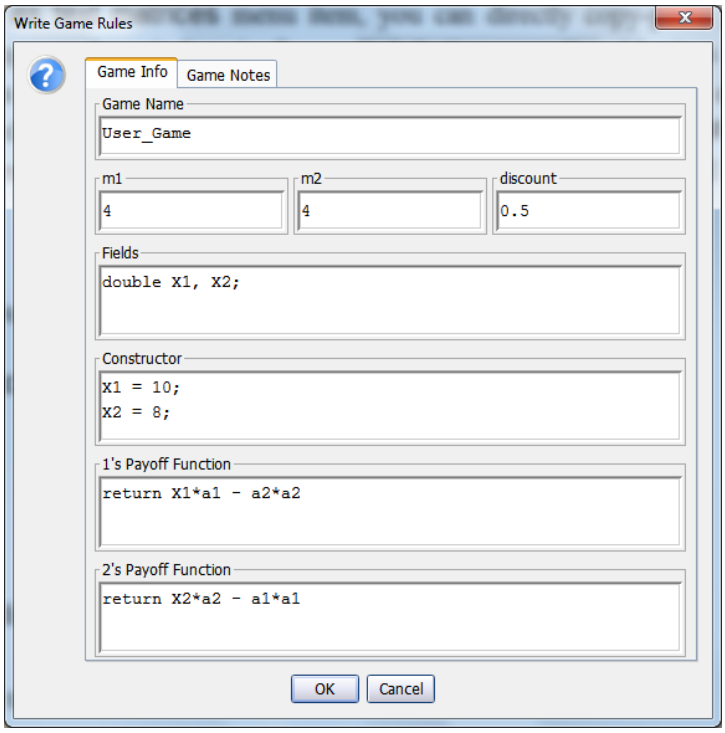


Figure 14: User-defined payoff rules

Due to some complicated issues involving classpaths and serialization, games generated in this fashion cannot be saved as `.rgm` files as they involve dynamically created and loaded Java classes that are thrown away after the end of the runtime session. They can only be saved in matrix form as `.mat` or `.txt` files, or as the special file format `.gmcode` which saves a concise representation of the source code representing the game, which can be recompiled by `rgsolve` in any runtime session.

## 4.3 File Types

`rgsolve` is compatible with four different file types for saving games.

- `.txt` files represent games as text data.
- `.rgm` files represent games in binary through Java's serialization functionality.
- `.mat` files represent games as MATLAB-readable binary files.
- `.gmcode` files represent games with user-defined Java-code rules defining payoffs.

We discuss each of these in turn below.

### 4.3.1 `.txt` games

The simplest way to store and load games for beginners is to save them as simple text files. Text files must be in the following format to be compatible with `rgsolve`:

```
<discount>
<m1>
<m2>
<G1>
<G2>
<actionsAreDisabled?>
<usableActions>
<Description>
```

`discount` is just the discount factor. $m_1$ and $m_2$ are the number of actions available to players 1 and 2. $G_1$ and $G_2$ are the payoff matrices; $G_i(a_1, a_2)$ should be the payoff to player $i$ when player $j$ plays $a_j$. Player 1 is the row player and Player 2 is the column player.

`actionsAreDisabled` should read either "`true`" or "`false`". If it reads `false`, `rgsolve` assumes that all action profiles in the game are usable in equilibrium. Otherwise, `rgsolve` expects an $m_1 \times m_2$ matrix of {`0`, `1`}, where "`0`" denotes that the action is not usable, and "`1`" denotes that the action is usable.

`Description` can be a String of any length containing game notes on the game. For example, a simple Prisoner's Dilemma with $\delta = .7$ could be written as in table 1. You can generate these text files in the program/environment of your choice. Matrices should have entries delimited by space characters, with each row on a new line.

| ***Prisoner's Dilemma with action*** $(1, 2) = (C, D)$ ***disabled:*** | ***Prisoner's Dilemma with all actions usable:*** |
|---|---|
| `0.7` | `0.7` |
| `2` | `2` |
| `2` | `2` |
| `2 -1` | `2 -1` |
| `3 1` | `3 1` |
| `2 3` | `2 3` |
| `-1 1` | `-1 1` |
| `true` | `false` |
| `1 0` | `The classic Prisoner's Dilemma` |
| `1 1` | |
| `The classic Prisoner's Dilemma` | |

Table 1: Prisoner's Dilemmas in `.txt` format

### 4.3.2 `.mat` games

This file type is compatible with MATLAB[7] To write a game that is readable by `rgsolve` in this format, in MATLAB generate

- $m_1 \times m_2$ matrices `G1` and `G2` in MATLAB storing the payoffs to player 1 and 2, respectively,
- a $1 \times 1$ matrix `delta` storing the discount rate,

[7] and with the earlier MATLAB `rgsolve` implementation.

- a $m_1 \times m_2$ matrix `usableAction` of {0,1} indicating which action profiles are usable (1) and unusable (2) (*optional*),
- A String `'notes'` holding any desired game description or notes (*optional*).

Then use the command

```
save('<gamename>.mat', 'G1', 'G2', 'delta', 'usableAction' (optional), 'notes' (optional));
```

to save the game as the file `<gamename>.mat`. For example, the code generating a Prisoner's Dilemma would be

```
G1 = [2 -1; 3 0];
G2 = G1';
delta = 0.7;
notes = 'A prisoner's dilemma.';
save('PD.mat', 'G1', 'G2', 'delta', 'notes');
```

#### 4.3.3 `.rgm` files

The easiest way to store games is to use Java's serialization capabilities. `.rgm` is a file extension specifically associated with serialized `rgsolve` game objects. This is probably the safest and most efficient way to store most games.

#### 4.3.4 `.gmcode` files

See section 4.2.2.

### 4.4 rgsolve's Predefined Games

Using the Predefined Games submenu in the Games menu, `rgsolve` can generate and load games for the user (using the `GameFactory` class).

**Cournot Duopoly** In this game, two oligopolists $i = 1, 2$ each produce a quantity $q_i$ and face a demand curve

$$P(q_1, q_2) = \max\{0, \alpha_0 - \alpha_1 (q_1 + q_2)\}$$

with constant marginal cost $c$. So profits are

$$\pi_i (q_1, q_2) = q_i [P(q_1, q_2) - c_i],$$

where $c_i$ is the constant marginal cost to $i$. If player $i$ has $m_i$ actions, then his action set is

$$\mathcal{A}_i = \left\{ \frac{k-1}{m_i - 1} \bar{Q}_i \mid k = 1, \ldots, m_i \right\},$$

where maximum output

$$\bar{Q}_i = \begin{cases} \max\left\{ \frac{\alpha_0}{\alpha_1}, \frac{\delta}{(1-\delta)^2} \frac{1}{c_i} \frac{(\alpha_0 - c_1)^2}{4\alpha_1} \right\}, & c_i > 0 \\ \frac{\alpha_0}{\alpha_1} & c_i = 0 \end{cases}.$$

**Bertrand Duopoly [Perfect Substitutes]** In this game, two oligopolists $i = 1, 2$ each charge a price $p_i$ and face a demand curve

$$Q_i(p_1, p_2) = \begin{cases} (\zeta_0 - \zeta_1 p_i)^+, & p_i < p_{-i} \\ \frac{1}{2}(\zeta_0 - \zeta_1 p_i)^+, & p_i = p_{-i} \\ 0, & p_i > p_{-i} \end{cases}.$$

Profits to $i$ are then

$$\pi_i(p_1, p_2) = Q_i(p_1, p_2)[p_i - c_i],$$

where $c_i$ is the constant marginal cost to $i$. If player $i$ has $m_i$ actions in the settings, then his actions in this game are prices

$$p_i \in \left\{ \frac{\zeta_0}{\zeta_1} \frac{k-1}{m_i - 1} \mid k = 1, \ldots, m_i \right\}.$$

**Bertrand Duopoly [Imperfect Substitutes]** In this game, two oligopolists $i = 1, 2$ each charge a price $p_i$ and face a demand curve

$$Q_i(p_1, p_2) = \max\left\{ a - b p_i + \frac{a + p_j}{b}, 0 \right\}.$$

Profits to $i$ are then

$$\pi_i(p_1, p_2) = Q_i(p_1, p_2)[p_i - c_i],$$

where $c_i$ is the constant marginal cost to $i$. If player $i$ has $m_i$ actions in the settings, then his actions in this game are prices

$$p_i \in \left\{ \frac{a}{b} \frac{k-1}{m_i - 1} \mid k = 1, \ldots, m_i \right\}.$$

**Random Normal Game** In this game, player $i$ draws a random $m_1 \times m_2$ payoff matrix $A^i$, where $A^i_{kh} \sim \mathcal{N}(\mu_i, \sigma_i^2)$ and

$$\operatorname{Corr}\left(A^1_{kh}, A^2_{kh}\right) = \rho \in [-1, 1].$$

**Random Uniform Game** In this game, player $i$ draws a random $m_1 \times m_2$ payoff matrix $A^i$ where $A^i_{kh} \sim_{\text{iid}} \mathrm{U}[0, 1]$.

**Prisoners' Dilemma** This is the classical prisoners' dilemma, with payoff bimatrix

| | **Cooperate** | **Defect** |
|---|---|---|
| **Cooperate** | $2, 2$ | $-1, 3$ |
| **Defect** | $3, -1$ | $1, 1$ |

Figure 15: Prisoners' Dilemma

**Hawk-Dove** This game has the payoff matrix

| | **Hawk** | **Dove** |
|---|---|---|
| **Hawk** | 0, 0 | 3, 1 |
| **Dove** | 1, 3 | 2, 2 |

Figure 16: Hawk-Dove

| | **L** | **M** | **H** |
|---|---|---|---|
| **L** | $16, 9$ | $3, 13$ | $0, 3$ |
| **M** | $21, 1$ | $10, 4$ | $-1, 0$ |
| **H** | $9, 0$ | $5, -4$ | $-5, -10$ |

Figure 17: Abreu-Sannikov Simple Example

**Abreu-Sannikov Example 1** This game is the example on page 10 of Abreu-Sannikov 2013: The default discount factor is $\delta = .3$.

**Abreu-Sannikov Example 2** This game is the example on page 18 of Abreu-Sannikov 2013:

| | **L** | **M** | **H** |
|---|---|---|---|
| **L** | 400, 530 | 0, -400 | 1, 1 |
| **M** | 1100, -1200 | 0, 0 | -400, 0 |
| **H** | 1, 1 | -1200, 1100 | 530, 400 |

Figure 18: Abreu-Sannikov Example 2

The default discount factor is $\delta = .6$.

# 5 The rgsolve API

### 5.0.1 Using the rgsolve package classes directly

To use the **rgsolve** package in your own Java projects (or projects that can access Java classes), you need to put the **rgsolve.jar** file on your classpath. This tells your JRE/JDK the location of the **rgsolve** classes and allows you to call them from any Java program on your machine. Settings the classpath varies from system to system, and your best bet is to search the internet for specific instructions for your system.

For example, on a Windows 7 machine, go to

Control Panel → System → Advanced System Settings → Advanced → Environment Variables

Under User Variables, click New, and enter

Variable Name: `classpath`
Variable Value: `.;< absolute location of rgsolve.jar >`

Alternatively, if you are using a Java IDE, you can simply add **rgsolve** to the build path of your project.

### 5.0.2 The rgsolve Javadocs

At the address

http://www.princeton.edu/~rkatzwer/rgsolve/doc/

one can find the API specification for the **rgsolve** package. Almost all of the classes and methods have Javadoc comments except for the GUI classes in the package **rgsolve.components**. Using this documentation, an experienced Java programmer should have no trouble using the **rgsolve** package in his own projects!

## 5.1 A simple example to get started!

Below is a simple example of some source code which calls the objects and methods of the **rgsolve** package directly.

```
package edu.princeton.repeatedgames.rgsolve.example;
import edu.princeton.repeatedgames.rgsolve.RGSolution;
import edu.princeton.repeatedgames.rgsolve.RGSolve;
import edu.princeton.repeatedgames.rgsolve.components.SolutionGraph;
import edu.princeton.repeatedgames.rgsolve.games.BimatrixGame;
import edu.princeton.repeatedgames.rgsolve.games.Game;


/**
 * A class with some simple examples to get a Java programmer
 * started with using the rgsolve package!
 */
public class RGSolveExample {

    public static void main(String[] args) {

        /* ---------------------------------- *
         * Example 1: The Prisoner's Dilemma  *
         * ---------------------------------- */
```

```
// Define payoffs and discount rate
double[][] G1 = { { 2, -1},
                  { 3,  1} };
double[][] G2 = { { 2,  3},
                  {-1,  1} };

double discount = .5;

// Instantiate the game
BimatrixGame pdgame = new BimatrixGame(
    G1, G2, discount, null, "The The Prisoner's Dilemma"
);

// Instantiate the solver for this game which will use
// the default parameters, and print output to the console
RGSolve rgsolve = new RGSolve(pdgame);

// Solve the game and store the solution
RGSolution pdsoln = rgsolve.solveGame();

// create a solution panel to view the solution
// to the game
SolutionGraph.showSolutionPanel(pdsoln);

/* ----------------------------------- *
 * Example 2: A Bertrand Game          *
 * ----------------------------------- */

// instantiate an anonymous class for our Bertrand game
int m1 = 100, m2 = 100;
double discount2 = .5;
final double A = 10, B = 2;  // Q = A - B * P
final double MC = .5;        // marginal cost

/**
 * A simple perfect-substitutes duopoly Bertrand game with
 * demand function Q = A - B * P and marginal cost MC
 */
Game bertrand = new Game(m1, m2, discount2, "100x100 Bertrand") {

    /** payoffs for player 1 */
    @Override
    public double payoff1(int a1, int a2) {
        double p1 = A / B * a1/(m1()-1);
        double p2 = A / B * a2/(m2()-1);
        if(p1 > p2)
            return 0;
        else {
            double Q = Math.max( A - B * p1, 0 );
            if(p1 == p2)
                return Q / 2 * (p1 - MC);
            else
                return Q * (p1 - MC);
```

```
                }
            }

            /** payoffs for player 2 */
            @Override
            public double payoff2(int a1, int a2) {
                double p1 = A / B * a1/(m1()-1);
                double p2 = A / B * a2/(m2()-1);
                if(p2 > p1)
                    return 0;
                else {
                    double Q = Math.max( A - B * p2, 0 );
                    if(p2 == p1)
                        return Q / 2 * (p2 - MC);
                    else
                        return Q * (p2 - MC);
                }
            }
        };

        // Instantiate the solver for this game which will use
        // the default parameters, and print output to the console
        rgsolve = new RGSolve(bertrand);

        // Solve the game and store the solution
        RGSolution bertrandSoln = rgsolve.solveGame();

        // create a solution panel to view the solution
        // to the game
        SolutionGraph.showSolutionPanel(bertrandSoln);
    }
}
```

# 6 The rgsolve MATLAB Wrapper

rgsolve is best utilized via the GUI launched from rgsolve.jar, or calling the package directly from your own Java project (as was discussed in section 5).[8] However, since MATLAB is a common programming enviroment in economics, rgsolve comes with some limited MATLAB capabilities. Including in the rgsolve.zip file are two MATLAB files:

```
JavaRGSolve.m
JavaRGSolveExample.m
```

The first file is a MATLAB class that encapsulates the rgsolve Java package. The second file is a MATLAB script giving an example of how to use the JavaRGSolve class. For these to work, these files must be in the same directory as the file rgsolve.jar, as they import this package into MATLAB. [9]

The best way to get a feel for these classes is to read the documentation included in the source code. To use them right away though, all you need to do is the following:

```
% Instantiate a solver object
solver = JavaRGSolve();
% Solve a game
solver.solve(G1, G2, delta);
```

The payoff matrices G1 and G2 should be double matrices of the same dimension, and the discount factor delta should be a scalar in $(0, 1)$. So for trivial example, we generate a random symmetric game

```
G1 = rand(5,5); G2 = G1'; delta=.8;
```

Calling the method solve(.) will construct a BimatrixGame Java object, pass it to the RGSolve Java class. The method returns an RGSolution Java object[10] and displays a GUI for exploring the game solution.

**From approximate to exact solutions.** The MATLAB wrapper *is* convenient for finding *exact* solutions for $V*$ (from approximate solutions, as described in Abreu-Sannikov 2013). We can simply call the command

```
% Get the exact solution using Matlab's symbolic solver
Vstar = solver.getExactSolution( solution )
```

If the MATLAB symbolic solver is succesful, the variable Vstar should contain an *exact* algebraic description of the extreme points of $V^*$.[11]

[8] Using rgsolve through MATLAB requires that games be stored as matrices. This may not be the most efficient way to store the games, particularly if payoffs are generated according to some rule (i.e. a Cournot game).

[9] Alternatively, one can use the command

```
javaaddpath('<path to rgsolve.jar>/rgsolve.jar');
```

to load the package if it is not in the same directory as your MATLAB application.

[10] See http://www.princeton.edu/~rkatzwer/rgsolve/doc/index.html

[11] MATLAB may fail and give a numerical solutions; the Mathematica equation solver is better for this in my experience.

# 7 Acknowledgments

I wrote all of the Java code in `rgsolve`, with the following exceptions:

- The `RowNumberTable` class written by Rob Camick and can be accessed here online at http://tips4java.wordpress.com/2008/11/18/row-number-table/.
(This provides the column of row-headers in the JTables of game payoffs.)

- Richard Katzwer adapted a C++ implementation of Andrew's Monotone Chain Algorithm from the website `The Algorithmist`, and the original C++ code can be found at http://www.algorithmist.com/index.php/Monotone_Chain_Convex_Hull.cpp.
The `rgsolve` version implements some robustness checks on the hull output in the flavor of the papers on robust 2D orientation problem papers by Katsuhisa Ozaki, Takeshi Ogita, Siegfried Rump, Shinichi Oishi, and others.

- The routines which allow games to be saved and opened as `.mat` files are thanks to classes in the JMatIO package, written by Wojciech Gradowski, and can be downloaded at http://sourceforge.net/projects/jmatio/ [12]

- The routines which allow user-defined payoff rules to be compiled and loaded onto the class path dynamically use code written by Morten Nobel, which can be found at http://blog.nobel-joergensen.com/2008/07/16/using-eclipse-compiler-to-create-dynamic-java-objects-2/
This itself uses the the free Eclipse compiler, `ecj.jar`, which can be found at http://download.eclipse.org/eclipse/downloads/ [13]

- The code which computes rational approximations to `double` is based in part on C code which can be found at http://shreevatsa.wordpress.com/2011/01/10/not-all-best-rational-approximations-are-the-convergents-of-the-continued-fraction/

- The look-and-feel of the `rgsolve` program is the TinyLaF Java look-and-feel, which was written by Hans Bickel and can be found at http://www.muntjak.de/hans/java/tinylaf/index.html [14]

- Some of the `rgsolve` plotting classes use JAMA: A Java Matrix Package, which can be found at http://math.nist.gov/javanumerics/jama/ [15]

[12] License: Copyright (c) 2006, Wojciech Gradkowski All rights reserved.
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
* Neither the name of the JMatIO nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

[13] License: `http://www.eclipse.org/legal/epl-v10.html`

[14] License: `http://www.gnu.org/licenses/lgpl.html`

[15] Copyright Notice: This software is a cooperative product of The MathWorks and the National Institute of Standards and Technology (NIST) which has been released to the public domain. Neither The MathWorks nor NIST assumes any responsibility whatsoever for its use by other parties, and makes no guarantees, expressed or implied, about its quality, reliability, or any other characteristic.

## References

Abreu, D., Pearce, D., and Stacchetti, E. (1990) "Toward a Theory of Discounted Repeated Games with Imperfect Monitoring". *Econometrica*, Vol. 58, pp. 1041-1063.
http://www.jstor.org/stable/2938299

Abreu, D., Sannikov, Y. (2013) "An Algorithm for Two Player Repeated Games with Perfect Monitoring". *Theoretical Economics* (Forthcoming)
http://econtheory.org/ojs/index.php/te/article/viewForthcomingFile/1302/8114/1