Interactively Solving Repeated Games: A Toolbox Version 0.2

Sebastian Kranz*

University of Cologne

March 27, 2012

Abstract

This paper shows how to use my free software toolbox *repgames* for R to analyze infinitely repeated games. Based on the results of Goldluecke & Kranz (2012), the toolbox allows to compute the set of pure strategy public perfect equilibrium payoffs and to find optimal equilibrium strategies for all discount factors in repeated games with perfect or imperfect public monitoring and monetary transfers. This paper explores various examples, including variants of repeated public goods games and different variants of repeated oligopolies, like oligopolies with many firms, multi-market contact or imperfect monitoring of sales. The paper also explores repeated games with infrequent external auditing and games were players can secretly manipulate signals. On the one hand the examples shall illustrate how the toolbox can be used and how our algorithms work in practice. On the other hand, the examples may in themselves provide some interesting economic and game theoretic insights.

*skranz@uni-bonn.de. I want to thank the Deutsche Forschungsgemeinschaft (DFG) through SFB-TR 15 for financial support.

Contents

1 Overview

A main goal of the theory of infinitely repeated games is to understand for different environments how and to which degree self-interested parties can sustain cooperative outcomes through repeated interaction. This tutorial describes a free software toolbox for R that allows to calculate optimal equilibria and the set of implementable payoffs for any discount factor in infinitely repeated games with imperfect or perfect monitoring monitoring that allow for monetary transfers. It is based on my joint paper with Susanne Goldlücke[1] (Goldlücke and Kranz, 2012) that develops and explains the underlying theoretical results. To understand how and why the software works and to interpret the results, I strongly recommend to have a look at our theoretical paper.

I believe that the presented toolbox can be quite helpful for economists and other social scientists who use game theoretic models to get a deeper understanding of many interesting social phenomena. Here are several situations, in which you may find the software useful:

1. You analyze a static game theoretic model of some economic phe-nomena and quickly want to check with some numerical examples the robustness of your predictions to repeated interactions (in particular, the robustness of your comparative statics).

2. Similarly, you may want to use the toolbox for robustness checks if you already analyze a repeated game but have considered only a particular class of strategies, like grim-trigger strategies.

3. You may want to check whether some particular class of equilibria you study are optimal or how far they are away from the boundary of optimal (pure strategy) equilibria. The toolbox only analyzes repeated games where monetary transfers are allowed, but the resulting payoff sets generate an upper bound to payoff sets of games where no monetary transfers are possible.

4. You can use the toolbox to develop and test conjectures about the structure of optimal equilibria in a particular repeated games, e.g. whether w.l.o.g. some symmetry constraints can be imposed. I try to illustrate this approach in several examples studied in this paper.

5. You want to design an experiment about repeated games (with monetary transfers) and want to quickly find the equilibrium payoff sets for different possible treatment configurations.

6. You may also consider the toolbox when teaching about repeated games, in particular if you think that students should not only learn math and economics, but should also acquire some programming skills. R is really a great software that is very popular among statisticians.

[1] Susanne just recently married and may still be better known by her former name Susanne Ohlendorf

Besides the core functions that solve a repeated game using our algorithm, the software contains several additional tools, in particular for graphical analysis, that facilitate testing and interpretation of the results and the analysis of comparative statics. So far, the toolbox itself is not yet well documented (the the source code is often commented, however) so this paper is the main source of documentation.

The main goals of this paper are the following:

1. Describe how my software toolbox for R can be used.

2. Make some advertisement for our theoretical paper by showing how our theoretical results can be applied.

3. Analyze some non-trivial examples of repeated games. Show how comparative statics of the payoff sets can be performed and how optimal strategies look like. I hope that the game theoretic and economic insights from some examples are in themselves interesting for some readers. I am planning to extend several examples and analyze them in separate papers.[2]

Section 2 explains how you can install the software. Afterward this paper is divided in two parts. Part I treats repeated games with perfect monitoring and Part II treats games with imperfect public monitoring. Each parts consist of different Sections that explain you how to use my toolbox (from basic to more advanced techniques) for different classes of games and at the same time contain some brief economic and game theoretic discussion of the results. In an appendix, I give some hints on debugging. (When I analyzed a new class of games, I nearly always made some mistakes in my code that specifies the stage game. I guess it is normal to need some debugging before everything runs fine). There are several exercises distributed across the different sections that invite you to explore my toolbox yourself.

2 Installation of the software

2.1 Installing R

My toolbox and all software you need to run it is open source and freely available for download.

First you have to install R, which is a very well developed software package for statistical and numerical computations with great graphic abilities, a very active community and a huge repository of contributed packages. To download the actual version of R visit the Comprehensive R Archive Network under

`http://cran.r-project.org/`

On the CRAN you can also find many tutorials that introduce you to R.

[2] I know that it is not good style to put half-finished work that has almost no literature review to the web. Still, I chose do to so, since I wanted to include a lot of examples in this paper to illustrate various ways how our algorithm and the toolbox can be applied.

2.2 Installing the package under Windows 32 bit

My toolbox is an R package with name *repgames*. To run it also requires the packages *skUtils*, *slam* and *glpkAPI*. My own packages repgames and skUtil are not yet available on the CRAN. If you use Windows 32 bit, you can download the zip files of the two packages from my homepage:

`http://www.wiwi.uni-bonn.de/kranz/`

You can install it in R by choosing the menu "packages" in your RGui and then the submenu "Install package from local zip files". Alternatively you can paste the current code, where you have to adjust the file path to the folder to which you have copied the zip files:

```
install.packages("D:/libraries/skUtils.zip", repos = NULL)
install.packages("D:/libraries/repgames.zip", repos = NULL)
```

Note: Use / instead of \ to specify file paths in R.

The two packages: "glpkAPI" and "slam" can be found on the CRAN. If you have internet access, you can simply type

```
install.packages("slam")
install.packages("glpkAPI")
```

which installs these packages directly from the CRAN. Before you can use the package repgames, you also have to load it in the actual R session. You can do this with the following command:

```
library(repgames)
```

(You do not have to manually load the other packages).

If installation fails, please send me an email.

2.3 Installing the package for a different operating system

If you want to use the package for a different operating system please send me an email.

2.4 Running Examples

In the different Sections below you find a series of examples. One way to try them out, is to copy-and-paste them directly from this PDF file. Alternatively, you can download to .r files on my homepage that contain the examples of Parts I and II of this paper.

2.5 Installing a text editor for R files

One convenient way to use R is to have some text editor, ideally with syntax highlighting for R files, and to copy-and-paste from the editor any sequence of R commands directly in the R console. Here is a list of editors for which syntax highlighting of R code is available:

`http://www.sciviews.org/_rgui/projects/Editors.html`

I am a Windows User and like Notepad++ very much. It comes with syntax highlighting for R.

`http://notepad-plus-plus.org/`

2.6 About RStudio

RStudio is a great IDE for R with integrated editor that can be used as replacement of the default RGui. I use RStudio for most of my work with R and generally would recommend to every beginner.

I would not recommend it for my packacke repgames, though. The reason is that the current version of RStudio (0.95) does not update plots during program execution on my computer. For most work that is not a big issue, but when solving longer models in the repgames package I use a lot of dynamic plots that give the user some impression what is going on. In the moment, this does not really work under RStudio. Thus, so far, I would recommend to use the default RGui when working with the repgames package.

Part I
Perfect Monitoring

3 A Simple Cournot Game

3.1 Initializing the game

As first example, let us consider the simple Cournot game from Abreu (1988) that is manually solved in Section 3 of Goldluecke & Kranz (2010). The payoff matrix of the stage game is given by

		Firm 2		
		L	M	H
Firm 1	L	10,10	3,15	0,7
	M	15,3	7,7	-4,5
	H	7,0	5,-4	-15,-15

We assume that every period consists of 3 stages: an ex-ante payment stage, an action stage, and an ex-post payment stage. In the payment stages the players can simultaneously conduct monetary transfers to each other and in the action stage they play the simultaneous move stage game specified above. We also allow for money burning (or equivalently for transfers to a non-involved third party). Payoffs are additive in money and stage game payoffs and players are risk-neutral. Payoffs in future periods are discounted by a discount factor $\delta \in [0, 1)$ and there is no discounting between the different stages within a period. For the moment, we assume that monitoring is perfect, i.e. all players perfectly observe all past action profiles.

I describe now how to initialize and solve the game using the `repgame` package and R. First, we have to initialize the payoff matrix for player 1 and 2:

```
# Define Payoff Matrices for player 1 and 2
               #L #M  #H
g1 = rbind(c( 10, 3, 0),    # L
           c( 15, 7, -4),   # M
           c(  7, 5,-15))   # H
g2 = t(g1)
```

You can simply copy and paste the code above into the RGui. To get help on an function in R simply type `?functionname`, e.g. `?rbind`. To look at a variable, you simply have to type its name into the command screen, e.g.

```
g2
```

```
##      [,1] [,2] [,3]
## [1,]   10   15    7
```

```
## [2,]    3    7    5
## [3,]    0   -4  -15
```

Note that in this PDF file, R output is typically commented out with ## before each output line. This is done to make it easier to copy and paste code into R. We now load the library and initialize the game:

```
# Load package repgames
library(repgames, warn.conflicts = FALSE)
```

```
## Loading required package: glpkAPI
## Warning message: package 'glpkAPI' was built under R version 2.14.2
## Loading required package: methods
## Loading required package: slam
## Loading required package: skUtils
```

```
# Initialize the model of the repeated game
m = init.game(g1 = g1, g2 = g2, lab.ai = c("L",
    "M", "H"), name = "Simple Cournot Game", symmetric = FALSE)
```

Ideally, you could get detailed information about the function init.game by typing `?init.game`. Practically, I have not yet written helpfiles for my toolbox. So most information is contained in this paper. While at some point of time the documentation hopefully extends, the reader interested in more details should look at the source code, which is partially commented.

Anyhow, the function `init.game` sets up basic information about the stage game and converts it into a format that is suitable for further calculation. If you wonder, why I set `symmetric=FALSE`, read this footnote.[3] All calculated objects are stored in the variable `m`. Type `m` to have a look at its content. You can access particular elements of `m` by the `$` operator. For example, type `m$g` in the R command line:

```
m$g
```

```
##     [,1] [,2]
## L|L   10   10
## L|M    3   15
## L|H    0    7
## M|L   15    3
## M|M    7    7
```

[3] If the stage game is symmetric and the flag `symmetric=TRUE` is set, the algorithm works a bit faster, since it only calculates optimal punishment profiles for player 1. But even though the game is symmetric, you can always specify to solve it without taken advantages of knowing that the game is symmetric. So why did I chose the later option by setting the flag `symmetric=FALSE`? Nothing important, just when looking at `m$opt.mat` (see below) the label of the action plans then contains a punishment profile for every player not only player 1. I found this nicer for an initial example.

```
## M|H   -4    5
## H|L    7    0
## H|M    5   -4
## H|H  -15  -15
```

The variable contains a matrix that stores the payoffs of the stage game: every row corresponds to an action profile and every column to a particular player. This is the standard format my package uses to store and process payoffs. There is a fixed scheme how action profiles are ordered that you probably can induce from the example. Action profiles are internally indexed by their row number in the matrix `m$g`. R has the convenient feature that rows and columns of a matrix can be accessed by specified names. For example,

```
m$g["H|M", ]
```

```
## [1]  5 -4
```

yields the payoff vector associated with the action profile (H, M). You would get the same result by typing `m$g[8,]`. When we called, `init.model` we provided with the parameter `lab.ai=c("L","M","H")`, a vector of action names and by default the names of action profiles have the structure $a_1|a_2|...|a_n$. Providing short sensible action names, is helpful when using the toolbox.

The next example, lists all pure strategy Nash equilibria of the stage game:

```
m$nash
```

```
## M|M
##   5
```

The variable `m$nash` is a numerical vector giving the index of the action profiles that are Nash equilibria. The first row is simply the named assigned to this profile.

3.2 Solving and plotting the game

Let us now solve the repeated game. We do this by pasting the following lines:

```
# Solve the game
m = solve.game(m, keep.only.opt.rows = TRUE)
```

I will explain further below the meaning of the parameter `keep.only.opt.rows`. If you take again a look at `m`, you will see that several new elements are added. The relevant information about the solution is stored in the matrix `m$opt.mat`.

```
# Show matrix with critical delta, optimal action plans
#   and payoffs
m$opt.mat
```

```
##                     delta  L Ue  V v1 v2 ae a1 a2   r UV opt
## (M|M),(M|M),(M|M) 0.0000  0 14 14  7  7  5  5  5 Inf  0   1
## (L|M),(M|H),(H|M) 0.2500  6 18  0  0  0  2  6  8   3 18   1
## (L|L),(M|H),(H|M) 0.3333 10 20  0  0  0  1  6  8   2 20   1
```

Each row characterizes an optimal action plan and payoff set for a certain interval of discount factors. For an explanation of what are optimal action plans and other important concepts for understanding the details of the solution, have a look at our theoretical paper (Goldluecke & Kranz, 2010).

The name of every row labels the optimal action plan (a^e, a^1, a^2) and the indices of the action profiles are given in the corresponding later columns. The action profile a^e is played in every period on the equilibrium path. The punishment profiles a^1 and a^2 for player 1 and 2, respectively, would be played for one period if a player once unilaterally defected from a required payment. The column `delta` denotes the lowest discount factor, for which the action plan is optimal and `r` is the corresponding maximal discount rate. Column `Ue` denotes the maximal joint payoff and `v1` and `v2` denote the punishment payoffs of player 1 and 2. `V` is simply the sum of all players' punishment payoffs.

The variables `L` denotes the highest liquidity requirement of all action profiles in the action plan. The liquidity requirement of an action profile (or action plan) plays a crucial role in our algorithm. If you just want to apply the software, you can ignore it, but I would recommend to look at our paper to learn more about it. Briefly said, to implement an action profile in a static problem with enforceable payments and liquidity constraints, the sum of all players liquidity must be at least equal to the liquidity requirement of the action profile.

The set of (pure strategy public perfect) equilibrium payoffs corresponding to a particular row is given by

$$\left\{ (u_1, ..., u_n) \mid \sum_{i=1}^{n} u_i \leq U^e \text{ and } u_i \geq v_i \text{ for all } i \right\}.$$

For example, if the discount factor satisfies $\delta \in [\frac{1}{4}, \frac{1}{3}]$(well, as usual 0.3333333 actually means $\frac{1}{3}$), the set of equilibrium payoffs is given by every payoff vector $u \in \mathbb{R}^2$ with$\sum_{i=1}^{n} u_i \leq 18$ and $u_i \geq 0$. On the equilibrium path, players infinitely often repeat the asymmetric equilibrium state action profile $a^e = (L, M)$. The punishment profiles of player 1 and 2 are $a^1 = (M, H)$and $a^2 = (H, M)$, respectively.

To plot maximal joint equilibrium payoffs U^e and lowest joint and payoffs for all discount factors, we can type

```
plot(m, xvar = "delta", yvar = c("Ue", "V"))
```

where the parameters should be self-explanatory. The result is shown in Figure 1.

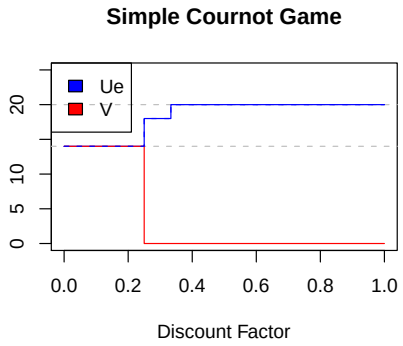


Figure 1: Maximal payoffs and sum of punishment payoffs of the repeated simple Cournot game (with side payments) for all discount factors.

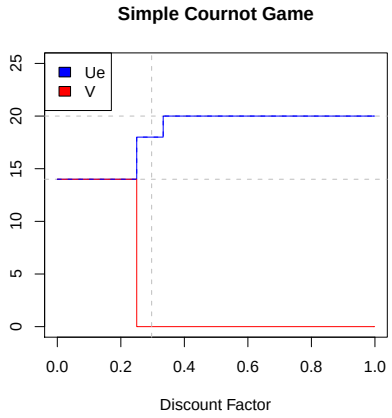
By default the plot shows the joint equilibrium and punishment payoffs as function of the discount factor, but the plot can be customized to show other payoffs. The dotted horizontal lines indicate the maximal joint payoffs and joint stage game payoffs. For repeated games with perfect monitoring (and finite action space and monetary transfers) the maximal implementable joint payoffs and lowest punishment payoffs are always step functions. Each step corresponds to another optimal action plan, i.e. to another row in `m$opt.mat`.

Identifying optimal action plans by clicking on the plot

Sometimes you will encounter some plot of payoffs and would quickly like to know about the optimal action plan that corresponds to some part of the plot. Insert the following code:

```
plot(m, xvar = "delta", yvar = c("Ue", "V"), identify = TRUE)
```

The flag `identify=TRUE` makes the plot interactive, i.e. you can left-click on the plot to get more information. I clicked on the second step of the Ue function, at the position indicated by the dotted grey vertical line in the figure below.



When clicking, the R console prints out the following information about the optimal action plan and payoffs for the selected level of δ:

```
"delta = 0.297"
                    delta L Ue V v1 v2 ae a1 a2 r   UV opt
(L|M),(M|H),(H|M) 0.297 6 18 0 0 0 2 6 8 2.367003 18 1
```

You can click on as many points as you like. To leave the interactive mode, you have to make a right-click on the plot and select the menu point stop. Alternatively, you can click on the red stop button in the toolbar of the RGui. I incorporated some interaction for most of the plotting functions you will encounter further below in this tutorial (e.g. `levelplot.payoff.compstat` or `plot.compare.models`). Just try out what happens if you call them with the parameter `identify=TRUE`.

Finally, let us see what happens if we call `solve.game` and set the flag `keep.only.opt.rows = FALSE`:

```
m = solve.game(m, keep.only.opt.rows = FALSE)
m$opt.mat
```

```
##                     delta  L Ue  V v1 v2 ae a1 a2   r UV opt
## (M|M),(M|M),(M|M) 0.0000  0 14 14  7  7  5  5  5 Inf  0   1
## (L|M),(M|M),(M|M) 0.5000  4 18 14  7  7  2  5  5   1  4   0
## (L|M),(M|H),(H|M) 0.2500  6 18  0  0  0  2  6  8   3 18   1
## (L|L),(M|H),(H|M) 0.3333 10 20  0  0  0  1  6  8   2 20   1
```

We find that `m$opt.mat` contains an additional 2nd row, for which the last column "opt" is set to 0. The second row corresponds to an action plan that optimizes all the static problems given total liquidity $L = 4$, but that is not an optimal action plan for any discount factor. By default, we have the option `keep.only.opt.rows=FALSE`, i.e. rows like the 2nd row will be kept.

3.3 Comparing optimal equilibria with equilibria in grim-trigger strategies

In many applied theoretical papers, repeated games are only analyzed by restricting attention to so called grim-trigger strategies. I implemented a function to calculate the payoff sets under grim-trigger equilibria. More precisely, I mean the payoffs that can be implemented by the class of equilibria that do not use monetary transfers and have a constant outcome path in the sense that a single action profile will be played in every period on the equilibrium path; any unilateral deviation of some player i is punished by reverting forever after to the worst Nash equilibrium of player i.

The following code solves the game by using only grim-trigger strategies:

```
# Solve the game with grim-trigger strategies
m.gt = set.to.grim.trigger(m)
m.gt = solve.game(m.gt)
m.gt$opt.mat
```

```
##     delta Ue  V   r ae delta1 delta2 g1 g2 L1 L2 L opt
## M|M 0.000 14 14 Inf  5  0.000  0.000  7  7  0  0 4   1
## L|L 0.625 20 14 0.6  1  0.625  0.625 10 10  5  5 5   1
```

The structure of the matrix is similar to that of `m$pm.opt.mat`. We find that when using only grim-trigger strategies the critical discount factor to sustain the fully collusive outcome (L, L) is much larger than for optimal strategy profiles: $0.625 = \frac{5}{8}$ compared to $\frac{1}{3}$. Playing the partial collusive outcome (L, M) on the equilibrium path is never optimal in the class of grim-trigger equilibria. We can also graphically compare the two models by typing:

```
plot.compare.models(m,m.gt,xvar="delta",yvar="Ue",
  legend.pos="topleft",m1.name = "opt", m2.name="grim")
```

The resulting output is shown in Figure 2:

4 Public Goods Games

4.1 Parametrizing two player public goods games

For a deeper analysis of some class of games, it is helpful to write functions that automatically initialize these games with a provided list of parameters. I want to illustrate an implementation with a simple two player public goods game. Each player i can choose a contribution level $x_i \in X = \{0, 1, ..., x_{max}\}$. Stage game payoffs shall be given by

$$g_i(x_i, x_j) = \frac{x_1 + x_2}{2} - k_i x_i \tag{1}$$

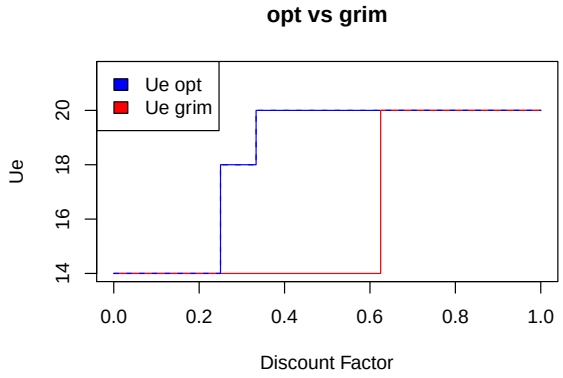


Figure 2: Comparison of payoffs under optimal equilibria and grim-trigger equilibria

where $k_i \in [\frac{1}{2}, 1]$ is a cost parameter that measures the marginal production cost of player i for each contributed unit. The following R code generates a function that initializes, solves and plots a parametrized public goods game in this class.

```
public.goods.game = function(X, k1,k2=k1) {
  # Two lines that are useful for debugging
  # (see hints below)
  store.objects("public.goods.game")
  # restore.objects("public.goods.game")

  # Initialize matrices of players contributions
  x1m = matrix(X,NROW(X),NROW(X),byrow=FALSE)
  x2m = matrix(X,NROW(X),NROW(X),byrow=TRUE)

  # Calculate payoff matrices for each player
  g1 = (x1m+x2m)/2 - k1*x1m
  g2 = (x1m+x2m)/2 - k2*x2m

  # Give the game a name
  name = paste("Public Goods Game k1=", k1, " k2=", k2, sep="")

  # Initialize the game
  m = init.game(n=2,g1=g1,g2=g2,name=name, lab.ai=X)

  # Solve and plot the model
  m = solve.game(m, keep.only.opt.rows=TRUE)
  plot(m)

  return(m)
}
```

Hints on R Programming

1. *Avoid loops and apply functions on vector or matrices whenever possible*

R is a great language, but awfully slow when it comes to loops. You should therefore always try to apply calculations on whole matrices or vectors whenever possible (by default R carries out all arithmetic operations element-wise). Creating some additional matrices like x1m and x2m, to facilitate calculations on matrices goes very quick in comparison.

2. *How can you follow step by step what my function does?*

You might want to understand step by step how my code within the function works. One method is to use the default debugger of R accessible e.g. through the `browser()` function. Since R is an interpreter you can, alternatively, simply paste the lines within the function into the R command screen and look at each step at the variables or plot them. The only problem is that the arguments of the function call, i.e. `X, k1,k2, name`, will not be properly assigned in the global environment. To solve this problem conveniently, I call within my function `store.objects()`, which stores a copy of all local objects in a global list under the provided name. The function `restore.objects` can be used the stored local objects from the last function call into the global environment. The following code explains, how I use this in practice:

```
# 1. Call the function with some parameters of interest
m = public.goods.game(X=0:2,k1=0.75)

# 2. Copy and paste the function code up to the
#    point of interest, starting with the
#    uncommented call to restore.object

restore.objects("public.goods.game")

# Initialize matrices of players contributions
x1m= matrix(X,NROW(X),NROW(X),byrow=!TRUE)
x2m = t(x1m)

# Calculate payoff matrices for each player
g1 = (x1m+x2m)/2 - k1*x1m
g2 = (x1m+x2m)/2 - k2*x2m
```

```
# 3. Inspect the variables of interest,e.g.
#    print or plot them.
# How do x1m and x2m look like?
x1m
```

```
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    1    1    1
## [3,]    2    2    2
```

```
x2m
```

```
##      [,1] [,2] [,3]
## [1,]    0    1    2
## [2,]    0    1    2
## [3,]    0    1    2
```

Of course, my method is a bit untidy in so far that I simply copy a lot of variables into the global environment, which perhaps may override some important global variables. However, in practice this has never been a big problem when I developed my toolbox and I found this way of debugging quite helpful to find where exactly something in my code went wrong.

3. A note for more advanced programmers

If the passed arguments are large, you might think that the function `store.objects` must be either quite time consuming, if it makes copies of each arguments, or flawed, if it just stores a pointer. In fact, it is not flawed but still relatively quick, since R uses a cool lazy evaluation feature. For the user, variables never behave like pointers, but internally R only creates a pointer when some object is copied. R makes an actual copy of an object only when it is changed. To understand what I mean, consider the following example:

```
x = numeric(10^8)          # Line 1
y=x                        # Line 2
y[1]=2                     # Line 3
y[1]=3                     # Line 4

c(x[1],y[1])               # Show start of both vectors
rm(x,y)                    # Removes x and y from memory
```

The first line creates a big vector filled with zeros and takes some time (if the vector is large enough compared to your computing power). The second line copies the vector to the variable y. The third and fourth line just change the first element of y. On first thought, one would suspect that line 3 and 4 are super fast while line 2 may take some time. In fact,

however, line 2 and 4 are super fast, while line 3 takes some time. You may even get an out of memory error in line 3. That is because the vector is physically copied only in line 3.

Let us us solve the public goods game with the parameters $X = \{0, 1, ..., 10\}$ and identical production cost of both players of $k_1 = k_2 = \frac{3}{4}$:

```
m = public.goods.game(X=1:10,k1=0.75,k2=0.75)
m$opt.mat
```

```
##                      delta   L  Ue   V   v1   v2  ae a1 a2   r  UV opt
## (1|1),(1|1),(1|1)      0.0 0.0 0.5 0.5 0.25 0.25   1  1  1 Inf 0.0   1
## (10|10),(1|1),(1|1)    0.5 4.5 5.0 0.5 0.25 0.25 100  1  1   1 4.5   1
```

We find that for all discount factors $\delta \geq \frac{1}{2}$, maximal contributions $(10, 10)$ can be sustained and for lower discount factors the only equilibrium is the Nash equilibrium of the stage game. Since the Nash equilibrium gives every player his min-max payoff of the stage game, it is always an optimal punishment profile. Given this fact and the symmetric structure, one might suspect that grim-trigger strategies can sustain full contribution for the same range of discount factor. You can verify this conjecture either by doing the math in your head or calculating the critical discount factor under grim-trigger strategies in the way explained in Section 3.3. Below, in Exercise 1, you are asked to investigate the more interesting case that the two players have different production costs.

4.2 Analyzing comparative statics

4.2.1 Symmetric costs

Before we come to the exercises, I want to illustrate how one can graphically analyze the comparative statics of the equilibrium set with respect to some parameter of the stage game, like the marginal production costs. The following little piece of code solves the model for a grid of different marginal production costs between $\frac{1}{2}$ and 1 and stores the solved models in the variable `m.list`.

```
pg.games = Vectorize(public.goods.game,
           vectorize.args = c("k1","k2"),SIMPLIFY=FALSE)

k.seq = seq(0.5,1,by = 0.01)
m.list = pg.games(X=0:10,k1=k.seq,k2=k.seq)
```

The first line transforms the function `public.goods.game` so that it can also take vector arguments for the cost parameters. It works as if you wold manually go through a loop and repeatedly apply the function `public.goods.game` for every element of the vector and store the results in a list. The next row specifies the vector of the different production costs that we are interested in. Here, we consider a grid with step size 0.01. The next line, calls the function and stores the resulting solved models in the variable `m.list` which is of the very flexible R type `list`.. To access, say the 50th solved model, simply type `m.list[[50]]`. During the execution of the code, you will see a little movie on the plot screen, since the function public.goods.game plots the payoffs for every solved model.

You can tell R to record all plots. To do this, click on the graphic window, then select from the menu "history" the element "Recording". From now on all future plots will be recorded. You can then go through the different plots with your page-up and page-down keys and investigate comparative statics in a flip-book fashion (try it out!).

If you paste the following line of code, you see a levelplot of maximal joint equilibrium payoffs as a function of the discount factor and the marginal production cost:

```
mat = levelplot.payoff.compstat(m.list, par = k.seq,
  xvar = "k", yvar = "delta", payoff.var="Ue",
  delta = seq(0,1,by=0.01), col.scheme = "grey")
```

The function `levelplot.payoff.compstat` returns a matrix with the joint equilibrium payoffs for every combination of the specified levels of `k` and `delta`. The resulting plot is shown in Figure 3:[4]

We find from the plot that the critical discount factor for which positive contribution levels can be sustained increases linearly in the production costs.

4.2.2 Asymmetric Costs

Let us now investigate asymmetric public goods games where the players have different production costs. Instead of immediately solving the model, you might find it more fun to make some quick conjectures of how the solutions will look like. For example, you could guess an answer to the following questions:

- Will players still either produce 0 or 10, or can intermediate production levels become optimal?
- Will grim-trigger equilibria still be able to implement the maximal joint payoffs for every discount factor or can optimal stationary equilibria with side payments implement strictly larger joint payoffs for some discount factors?

[4]The figures level plots will actually look better if you paste the code yourself. I have not yet found out how to get rid of these annoying white lines in the plot and legend when converting the levelplots into eps files.

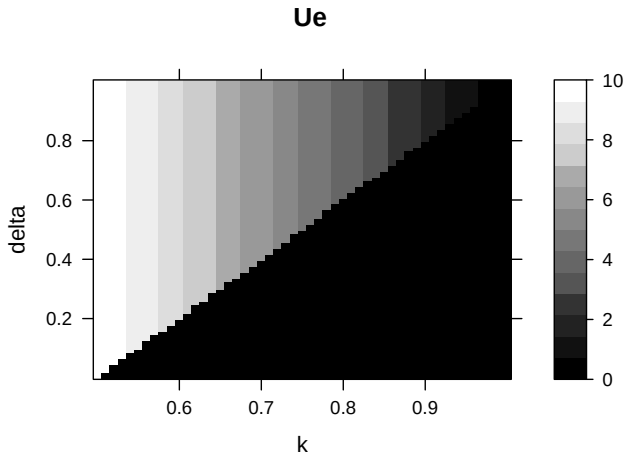


Figure 3: Comparative statics of maximal joint payoffs w.r.t. δ and symmetric production costs k. Darker points correspond to lower joint payoffs U^e.

- Does the maximal payoff only depend on the sum of production costs $k_1 + k_2$, or only on the minimum production costs $min(k_1, k_2)$, or is the structure more complicated?

You can then check whether the computed examples are consistent with your guesses or provide counter examples. To look at a single example, say $k_1 = 0.6$ and $k_2 = 0.7$, you can type:

```
m = public.goods.game(X=1:10,k1=0.6,k2=0.7)
m.grim = solve.game(set.to.grim.trigger(m))
m.grim$opt.mat
```

```
##        delta  Ue   V     r  ae delta1 delta2  g1  g2  L1  L2   L opt
## 1|1   0.0000 0.7 0.7   Inf   1 0.0000 0.0000 0.4 0.3 0.0 0.0 1.8   1
## 8|6   0.2857 5.0 0.7 2.500  76 0.2800 0.2857 2.2 2.8 0.7 1.0 0.4   1
## 9|7   0.3000 5.7 0.7 2.333  87 0.2667 0.3000 2.6 3.1 0.8 1.2 0.2   1
## 10|7  0.3000 6.1 0.7 2.333  97 0.3000 0.2667 2.5 3.6 0.9 1.2 1.2   1
## 10|8  0.3111 6.4 0.7 2.214  98 0.2571 0.3111 3.0 3.4 0.9 1.4 0.6   1
## 10|9  0.3556 6.7 0.7 1.813  99 0.2250 0.3556 3.5 3.2 0.9 1.6 0.2   1
## 10|10 0.4000 7.0 0.7 1.500 100 0.2000 0.4000 4.0 3.0 0.9 1.8 0.0   1
```

```
m$opt.mat
```

```
##                       delta   L  Ue   V  v1  v2  ae a1 a2     r  UV opt
## (1|1),(1|1),(1|1)    0.0000 0.0 0.7 0.7 0.4 0.3   1  1  1   Inf 0.0   1
## (10|1),(1|1),(1|1)   0.2000 0.9 4.3 0.7 0.4 0.3  91  1  1 4.000 3.6   1
## (10|2),(1|1),(1|1)   0.2200 1.1 4.6 0.7 0.4 0.3  92  1  1 3.545 3.9   1
## (10|3),(1|1),(1|1)   0.2364 1.3 4.9 0.7 0.4 0.3  93  1  1 3.231 4.2   1
## (10|4),(1|1),(1|1)   0.2500 1.5 5.2 0.7 0.4 0.3  94  1  1 3.000 4.5   1
## (10|5),(1|1),(1|1)   0.2615 1.7 5.5 0.7 0.4 0.3  95  1  1 2.824 4.8   1
```

```
## (10|6),(1|1),(1|1)  0.2714 1.9 5.8 0.7 0.4 0.3  96  1  1 2.684 5.1   1
## (10|7),(1|1),(1|1)  0.2800 2.1 6.1 0.7 0.4 0.3  97  1  1 2.571 5.4   1
## (10|8),(1|1),(1|1)  0.2875 2.3 6.4 0.7 0.4 0.3  98  1  1 2.478 5.7   1
## (10|9),(1|1),(1|1)  0.2941 2.5 6.7 0.7 0.4 0.3  99  1  1 2.400 6.0   1
## (10|10),(1|1),(1|1) 0.3000 2.7 7.0 0.7 0.4 0.3 100  1  1 2.333 6.3   1
```

From the results you will already get some answers to the first two questions above and may generally gain a better intuition of the structure of the game. The main factors that drive the results is that, given an identical contribution level, the low cost player has lower incentives to deviate than the high cost player. Furthermore, under optimal equilibria with monetary transfers, the high cost player can pay money to the low cost player in return for high contribution levels.

We can also perform graphical comparative statics with respect to the two cost parameters. The following code produces the levelplot shown in Figure 4:

```
# Generate grid of k1 and k2 combinations between 0.5 and 1
k.seq = seq(0.5,1,by = 0.02)
# (make.grid.matrix is an own function similar to expand.grid)
k.grid = make.grid.matrix(x=k.seq,n=2)
colnames(k.grid)=c("k1","k2")

# Solve the model for all combinations of k1 and k2
m.list = pg.games(X=0:10,k1=k.grid[,1],k2=k.grid[,2])

# Show level plot of payoffs for delta=0.5
delta = 0.5
levelplot.payoff.compstat(m.list, par = k.grid ,
  xvar = "k1", yvar = "k2", payoff.var="Ue",
  delta = delta, col.scheme = "grey")
```

.

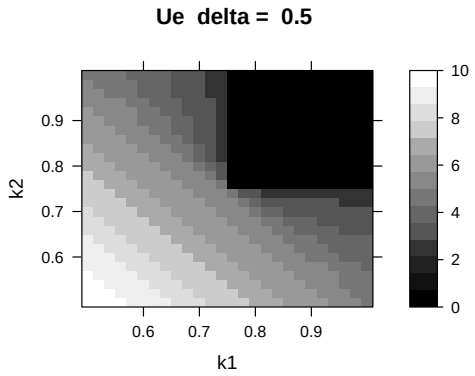


Figure 4: Payoff comparative statics in costs of the two players

For low levels of production costs k_1 and k_2, the iso-profit curves are linear, i.e. total payoffs only depend on the sum of production costs $k_1 + k_2$. This result makes perfect sense, since if both production costs are low, players do not have strong incentives to deviate and optimally choose full contribution $x_1 = x_2 = 10$.

For larger production costs, iso-profit curves are no longer linear and the cost of the low cost player is more decisive for total profits. This makes intuitive sense in so far that it is optimal to shift public good provision to the low cost player who may receive a monetary reimbursement from the high cost player. If production costs become too large that they fall into the black area on the top right, no contributions can be sustained.

To generate a flip-book of these plots for different levels of δ, turn on the graphic recording in R, paste the code below and finally click on the plot window and go for- and backward with the arrow keys:

```
for (delta in seq(0,1,by=0.1)) {
        levelplot.payoff.compstat(m.list, par = k.grid,
      xvar = "k1", yvar = "k2", payoff.var="Ue",
      delta = delta, col.scheme = "grey")
}
```

4.3 Analyzing n-player games

While so far we only investigated games with $n = 2$ players, the toolbox can also analyze repeated games with more players. While for 2 players it is convenient to specify the stage game payoffs by simply passing the payoff matrix of both players to the function `init.model`, there is a different way to specify payoffs for general n-player games. Let us first look at an example:

```
# Creates a n-player public goods game
# n is the number of players.
# X is the set of different contribution levels
# k is a n x 1 vector with production costs for every player

pg.game = function(n,X,k=rep(((1+1/n)/2),n)) {
  # A function that returns a payoff matrix
  # given a action matrix x.mat, of which
  # every row corresponds to one action profile
  g.fun = function(x.mat) {
    g = matrix(0,NROW(x.mat),n)
    xsum = rowSums(x.mat)
    for (i in 1:n) {
      g[,i] = xsum / n - k[i]*x.mat[,i]
    }
    g
```

```
  }
  name=paste(n,"Player Public Goods Game")
  m = init.game(n=n,g.fun=g.fun,action.val = X,
                      name=name, lab.ai=round(X,2))
  m=solve.game(m)
  plot(m)
  m
}

# Solve a 3 player public goods game with
# asymmetric production costs
m = pg.game(n=3,X=0:10,k=c(0.4,0.6,0.8))
```

Instead of passing payoff matrices to the function `init.game`, we pass a user defined function `g.fun` and a vector (or list of vectors for every player) `action.val`. The parameter `action.val` simply associates to every action of a player some number. Here it will simply be the contribution level. The function `g.fun` takes a vector of action profiles (i.e. a matrix of action values, where every row corresponds to an action profile) and returns a corresponding matrix of stage game payoffs.

Advanced Programming Hint:

When I define the function `g.fun` within the function `pg.game`, I use in the function body of `g.fun` the local variables `n` and `k` from `pg.game`. R saves information in which environment the function `g.fun` was created and when `g.fun` is called somewhere later, it will look up its variables from that original environment. The local environment information will be retained even after the function `pg.game` is exited. To see how it works, consider the following code example:

```
make.f = function(i,j) {
  f = function() {
    i
  }
  i=j
  return(f)
}
f = make.f(i=1,j=2)
g = function() {
  i=3
  return(f())
}
i=4
# What will be the output of the following call?
g()

## [1] 2
```

Any guess what the call `g()` will return? It will be 2. The function `f` looks up in in the local environment associated with the call to `make.f` where `f` was created. In that local environment `i` took the value of `j`, i.e. 2, before we exited `make.f`.

Note that the number of action profiles grows exponentially with the number of players. While the software should handle perfect monitoring games up to a million action profiles well and quickly on normal PCs, a too large number of action profiles can lead to problems, like running out of memory or excessive computation time. For approximating continuous stage games or games with a large number of players, you should use the approach presented in Sections 5 and 6.

Before concluding this subsection, let us have a look at the joint equilibrium payoffs as function of total liquidity:

```
plot(m,xvar="L",identify=TRUE)
```

We can see in Figure 5, that $\bar{U}^e(L)$ is a concave piece-wise linear function in L. You probably have some conjecture why we have these kinks in $\bar{U}^e(L)$. You can strengthen your intuition by using left-clicks to interactively explore the plot.

4.4 Exercises

The two exercises below ask you to explore some variations of repeated public goods games by modifying the examples described above.

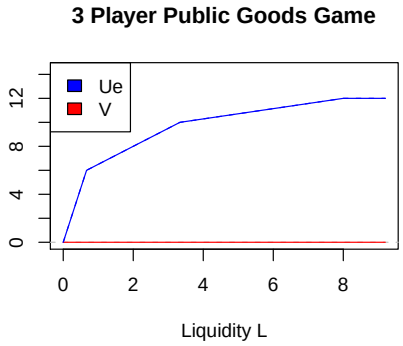


Figure 5: Public goods game (3 players with asymmetric costs)

Exercise 4.1. More Comparative Statics for Public Goods Games In the experimental literature, linear public goods game are often parametrized in a different way than presented above. Cost of contributing one unit is typically normalized to one and key parameters are the marginal per capita return and the marginal total return of one contributed unit. Isaak & Walker (1988) and others found the following stylized facts from economic experiments on finitely repeated public goods games:

- Ceteris-paribus, the level of contributions increases in the marginal per capita return. In other words, keeping the marginal total return constant, average contribution levels typically fall in the number of subjects.
- There is also weaker evidence that average contributions increase if the marginal per capita return is kept constant, but the number of subjects and with it the total return of contributions increase.

Reparametrize the public goods game using marginal per capita return, number of players and marginal total return as possible parameters (of course only 2 of them will be actually passed to the function). Analyze the comparative statics of contribution levels (or critical discount factors) with respect to the different parameters. Are they roughly in line with the experimental stylized facts from finitely repeated public goods games? Compare your comparative statics with those of your most favorite model of social preferences. If you should not yet have chosen a most favorite social preference model, have a look at Kranz (2010)! :)

Exercise 4.2. Public Goods Games with a Costly Punishment Technology Write a new function that allows to solve repeated public goods games, in which players can assign costly reduction points to the other player to reduce his payoffs. Stage game payoffs shall be given by

$$g_i(x, r) = \frac{x_1 + x_2}{2} - k_i x_i - r_j - \gamma_i r_i \tag{2}$$

where r_i is the number reduction points that player i assigns to player j and $\gamma_i > 0$ shall be a parameter that measures the cost of assigning one reduction

point. Reduction points r_i and contribution levels x_i must be chosen from some finite set. (Hint: The action space becomes simpler, if you assume that in a given period players can either contribute to the public good or assign reduction points, but cannot do both at the same time.).

In economic experiments on finitely repeated (or one shot) public goods games, it often turns out that augmenting the public goods game by a punishment technology substantially increase average contribution levels

Answer the following questions: Can the inclusion of such a punishment technology also increase the maximal sustainable contribution levels in infinitely repeated games? What if we would not calculate optimal equilibria, but only restricted attention to grim-trigger strategies?

Also perform some comparative statics with respect to the different parameters of the game.

As extension, analyze public goods games with three player and compare punishment technologies that can target reduction points on specific players with anonymous punishment technologies that always distribute reduction points equally.

5 Approximating Continuous Cournot Models

In this Section I want to illustrate, how for the case of perfect monitoring the toolbox can handle fine approximations of stage games with continuous action spaces. In Subsection 5.3, I will also illustrate, how one can handle games where some players have a multidimensional action spaces (e.g. selling multiple products).

5.1 Approximating a Cournot oligopoly with the known methods

The stage game shall be a symmetric linear Cournot model with n firms, who face an inverse linear demand function of the form

$$P(Q) = A - BQ.$$

All firms shall have identical constant marginal costs MC.

One way to approximate this game is simply to specify a grid of action profiles and use the method illustrated in Section 4.3. The following code initializes and solves a two player Cournot game of this class with $A = 100$, $B = 1$ and $MC = 10$:

```
init.cournot = function(n=2,q.seq, A, B, MC) {
  P.fun = function(Q) {A-B*Q}
  cost.fun = function(q) {MC*q}
  g.fun = function(qm) {
```

```
    Qm = rowSums(qm)
    Pm = P.fun(Qm)
    Pm[Pm<0]=0
    g = matrix(NA,NROW(qm),n)
    for (i in 1:n) {
      g[,i] = Pm*qm[,i]-cost.fun(qm[,i])
    }
    g
  }
  name=paste(n,"Cournot Game (g.fun)")
  m = init.game(n=2,g.fun=g.fun,action.val = q.seq,
                name=name, lab.ai=round(q.seq,2))
  return(m)
}

# Parameters of the model
n=2; A=100; B=1; MC=10;
q.seq = seq(0,100,by=1); # Grid of possible quantities
m = init.cournot(n=n,q.seq=q.seq,A=A,B=B,MC=MC)
m = solve.game(m)
plot(m,legend.pos = "right")
```

The models is solved very quickly and you see the resulting payoffs for shown in Figure 6.

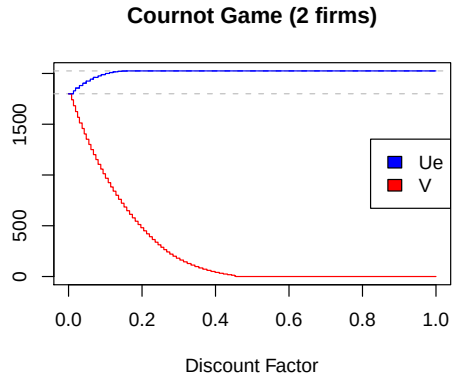


Figure 6: Payoffs for discretized Cournot Duopoly

We find that firms can perfectly collude already for very low discount factors. (Section 8, shows how monitoring imperfections make collusion much harder...).

Now let us analyze what happens if there are more firms in the market. Set the parameter $n = 4$, init and solve the model. Does your program run without error and in reasonable time? Then congratulations to your computer's memory... on my computer the program refuses to run and throws an out of memory error.

Basically, with 4 firms and 100 actions per firm, the discretized version of the stage game has $100^4 = 100$ million action profiles. My computer cannot solve it, because the underlying algorithm used for solving the game is not very memory efficient. It generates a matrix of all possible action profiles and manipulates it.

Besides the issue of practical computation there is also a small theoretical issue. The algorithm used above calculates cheating payoffs by considering only the action profiles from the specified grid, cheating payoffs in the continuous version of the stage game might be higher, however. As result, it is not clear whether the resulting payoff set of the discretized game is a lower or upper approximation of the payoff set of the continuous game.

5.2 Alternative ways to solve models with continuous action spaces

The toolbox contains alternative procedures for games with perfect monitoring, which facilitate the analysis fine discretizations of multidimensional continuous stage games and guarantee that the computed payoff sets are always a lower approximation of the payoff sets of the continuous game. These procedures can be recognized by the prefix `pmx`.

The procedures require that in addition to the stage game payoffs $g(a)$, you also have to manually provide functions that calculate cheating payoffs $c(a)$ for any possibly relevant[5] action profile of the continuous stage game $a \in A$.[6]

In the Cournot game above, we know that firm i's best-reply function is given by

$$q_i^*(Q_{-i}) = max\left\{\frac{A - MC}{2B} - \frac{1}{2}Q_{-i}, 0\right\}$$

The corresponding cheating payoffs are given by

$$c(q) = \begin{cases} \frac{(A-MC-BQ_{-i})^2}{4B} & \text{if } A - MC - BQ_{-i} > 0 \\ 0 & \text{otherwise} \end{cases}$$

Generally, knowing the payoff function $g(a)$ and cheating payoff function $c(a)$, we can calculate the liquidity requirement $L(a)$, joint payoffs $G(a)$ and player i's cheating payoffs for any sampled action profile a. To compute inner approximations of the sets of SPE payoffs, we can draw a finite random sample of action profiles in order to calculate lower bounds of the functions $\bar{U}^e(L)$ and $\bar{v}^i(L)$ for the continuous game. As the sample size grows large, these lower bounds converge in probability to the true functions and the corresponding payoff sets converge from the inside to the true payoff sets of the continuous stage game.

[5] If on theoretical grounds, we can rule out some action profiles as candidates for optimal action profiles, we also do not have to provide cheating payoffs (see Section 6, for an example)

[6] In principle, one could obtain an approximation of the general function $c(a)$ by telling R to perform numeric optimization using the stage game payoff function $g(a)$. However, I fear that this would be quite slow, i.e. closed form solutions to $c(a)$ are preferable.

The practical issue is to sample action profiles in a way that achieves relatively quick convergence for most stage games. I implemented so far two approaches that can each be parametrized in different ways. The first approach is a very simple adaptive grid refinement method that does not rely heavily on random sampling, the second approach uses more random sampling.

Before discussing some more details of the algorithms, let me show how to use them. Using the alternative methods requires to initialize the stage game with the function `pmx.init.game`, instead of using `init.game`. The following code initializes and solves the Cournot game for 4 players using the new methods:

```
pmx.init.cournot = function(n=2, q.range, A, B, MC) {
  # First part
  store.objects("pmx.init.cournot")
  #restore.objects("pmx.init.cournot")

  # Stage game payoffs
  gx.fun = function(q) {
    Q = rowSums(q)
    P = A-B*Q
    P[P<0]=0
    g = matrix(NA,NROW(q),n)
    for (i in 1:n) {
      g[,i] = (P-MC)*q[,i]
    }
    g
  }

  # Stage game cheating payoffs
  cx.fun = function(q) {
    Q   = rowSums(q)
    c.mat = matrix(NA,NROW(q),n)
    for (i in 1:n) {
      Q_i = Q-q[,i]
      c.mat[,i] = (A-MC-B*(Q-q[,i]))^2 / (4*B)
      c.mat[A-MC-B*(Q-q[,i])<0,i] = 0
    }
    c.mat
  }

  # Second part
  name=paste("Cournot Game (",n," firms)",sep="")
  m = pmx.init.game(n=n,nx=n,x.range=q.range,symmetric=TRUE,
        gx.fun=gx.fun,cx.fun=cx.fun,
        name=name, names.x=paste("q",1:n,sep=""))
  return(m)
```

```
}
# Parameters of the model
n=4; A=100; B=1; MC=10;
q.max = A / (n-1)   # maximal output a firm can choose
m = pmx.init.cournot(n=n,q.range=c(0,q.max),A=A,B=B,MC=MC)
```

Take a look at the line m = pmx.init.game(...). The parameter n denotes, as usual, the number of players. We have a new parameter nx, which denotes the number of "activities". An activity is some real number that influences stage game payoffs and a players' action may consist of one or several activities. In the Cournot example every player decides only on a single activity: the output in the market. This means actions and activities are identical. Below, we will explore a model of multi-market collusion, where every firm's action can consist of several activities. The parameter x.range specifies the intervals of the real line out of which activities have to be selected. If can either by a $nx \times 2$ matrix, where each row specifies separate ranges for every activity, or if every activity shall have the same range, the x.range can be given by a simple tuple (x.min,x.max). Here the minimal output of each firm shall be 0 and the maximal output $A/(n-1)$. This means $n-1$ firms are able to produce a total output that yields to a market price of zero. You can choose a larger bound if you like.

The parameter gx.fun specifies a function that returns a $T \times n$ dimensional matrix of payoffs given an $T \times nx$ matrix of activities, in which each row corresponds to one activity profile. Similarly, the function cx.fun must return a matrix of cheating payoffs for every player given a matrix of activities. Finally, the parameter names.x is a vector that specifies a label for each activity.

The following code solves the model using the adaptive grid refinement method:

```
# Solve with adaptive grid refinement
cnt = list(method="grid",
           num.step.start = 8, num.step.end = 128 ,
           step.size.factor = 4, grid.size = 100000)
m.grid = solve.game(m,cnt=cnt)
```

Solving the model takes, some time but we considered a relatively fine approximation. The solution method and corresponding control parameters are specified in a list `cnt`. The adaptive grid refinement method and the meaning of the parameters is explained in the box below (a slightly different parametrization is given in Section 6).

The adaptive grid refinement method

The algorithm performs the following procedure for every state $k = e, 1, ...n$, to approximate the functions $\bar{U}^e(L)$ and $\bar{v}^i(L)$. I will explain it for the equilibrium state. Basically, the algorithm starts with a coarse grid

of action profiles and notes which action profiles are optimal on that coarse grid. The parameter `num.step.start = 8`, specifies that the initial grid shall consists of 8 levels per activity, i.e. the initial grid contains $8^4 = 4096$ different activity profiles. For all of them, we calculate $G(a)$ and $L(a)$ and keep those profiles $\hat{A}$ that are optimal in the sense that for no $\hat{a} \in \hat{A}$ there is some other profile a on the coarse grid with $L(a) \leq L(\hat{a})$ and $G(a) \geq G(\hat{a})$ with one of the inequalities holding strictly. These action profiles can be used to generate a first approximation to the function $\bar{U}^e(L)$.

In the next step, we refine the grid by dividing the step size by the specified parameter step.size.factor. Instead of examining every activity profile in the global grid over the whole activity range, we examine local grids around those points that were optimal in the coarse grid. So basically, I take every point that was optimal and examine a `nx` dimensional local grid with that point in the center. The maximum size of the local grid is given by the parameter `grid.size`. For the given maximal grid.size of 100000 and a 4 dimensional activity space, the local grid extends

$$floor\left[\frac{1}{2}\left(\frac{log(100000)}{log(4)} - 1\right)\right] = 3$$

steps in every direction. Points in the local grid that are optimal, will be added to $\hat{A}$ and those that cease to be optimal will be removed. If we find an optimal point at the boundary of a local grid, we will also explore a local grid around that point.

The algorithm proceeds in this fashion until the number of grid points per activity rises above the parameter num.step.end.

One can also set a flag `use.random.start.points = TRUE`. One then uses a variant of the algorithm in which the initial grid is not fully explored, but a random sample of points (by default 50000) is drawn and the algorithm proceeds with the optimal points from this sample. The advantage is that one can start with finer initial grids.

To give the user a feeling of the convergence process, I plot the actual approximations to the functions $\bar{U}^e(L)$ and $\bar{v}^i(L)$ during the solution process.

Let us solve the model again using the random sampling algorithm:

```
# Solve with random sampling
cnt = list(method="random", step.size=0.1,
           size.draw = 1000,num.draws = 200,
           local.abs.width=4, prob.draw.global = 0.3)
m.ran = solve.game(m,cnt=cnt)
plot(m.ran)
```

The basic idea of the algorithm is explained in the box below.

About the random sampling method

Basically, we could simple sample activity profiles by drawing activities independent from each other and calculate the resulting approximations for $\bar{U}^e(L)$ and $\bar{v}^i(L)$ from the drawn activity profiles. If we draw sufficient many profiles, we will converge in probability towards the true functions. There are several ways to improve small sample performance, however.

My algorithm does the following:

The algorithm still uses a combination of local and global grid search. With probability prob.draw.global a random sample of activity profiles is drawn from the global grid and with probability 1-prob.draw.global from a local grid around some optimal point (each edge of the local hypercube of size local.abs.width). Each random sample has size size.draw and a total of num.draws samples is drawn.

The crucial factor of the algorithm is that the separate activities that form an action profile are not always independently drawn from each other. For a small number of activities this would not matter much, but below I will apply the algorithm also on a Cournot model with 100 firms. If the number of activities nx grows large, we have the problem that by a law of large numbers, the drawn activity profiles are very likely to look quite similar in some dimensions. For example, the average output across 100 firms would be unlikely to deviate much from the mean of the specified range. e.g. with respect to the average activity level within the profile. So it is helpful to draw from distributions that allow for positive or negative correlations between subsets of activities. I implemented a very simple procedure that does this job, which you can look up in the source code of the function pmx.draw.x.mat.

Another issue is how to select the points, around which the local grid shall be extended. Simply choosing randomly from all optimal points, might have the effect that the grid gets refined only at some particular location, where in each refinement step a lot of new optimal points are selected. To draw more evenly, I give precedence to those optimal points that were drawn from the global grid.[7]

A more even search is also facilitated by the fact that I still draw from a finite grid, with size given by the parameter step.size. I never draw a local grid twice from some optimal point when there are still some optimal points left, from which I have not yet drawn a local grid.

Again you get a graphical illustration of the convergence process. The following code compares the approximated $\bar{U}^e(L)$ from the two solution methods:

```
# Compare the two models graphically
plot.compare.models(m.grid,m.ran, xvar="delta", yvar="Ue",
                    m1.name="grid", m2.name="random",
                    identify = TRUE, legend.pos="right")
```

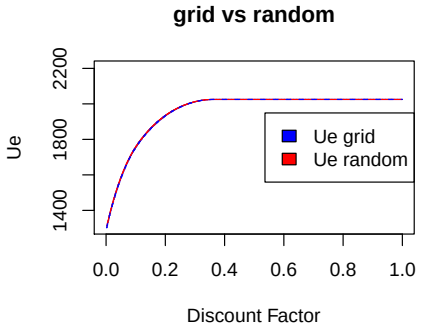


Figure 7: Comparing solutions of the two methods

The resulting plot is shown in Figure 7 below

We find that the two methods yield almost the same solutions, which is quite encouraging. The grid refinement method becomes quickly infeasible as the dimensionality of the action space increases, while the random sampling mechanism still tends to do a good job. Using the function call

```
m = refine.solution(m, cnt = cnt)
```

you can apply a different solution method described in `cnt` on a game in order to refine a previous solution. You can refine a solution as often as you like.

To test the toolbox for $n = 100$ firms, call:

```
# A hundred 100 firms
# Parameters of the model
n=100; A=100; B=1; MC=10;
q.max = A / (n-1)   # maximal output a firm can choose
m = pmx.init.cournot(n=n,q.range=c(0,q.max),A=A,B=B,MC=MC)
cnt.e = list(method="random", step.size=0.05,
          local.abs.width = 0.5, prob.draw.global = 0.3,
          size.draw = 1000,num.draws = 100)
cnt.i = cnt.e
cnt.i$num.draws = 400
m.ran.100 = solve.game(m,cnt=cnt.e, cnt.i=cnt.i)
plot(m.ran.100)
```

I chose to pick more samples for the characterization of the punishment state (cnt.i), since the graphical representation suggested that convergence for the punishment state went slower. Below, we will check how well our approximation for the 100 firms Cournot game worked.

5.3 Finding and imposing restrictions on the structure of optimal action profiles

If for a problem at hand, we have some theoretical results about the structure of optimal action profiles, computational complexity may be substantially reduced. In particular, knowledge that w.l.o.g. we can impose some symmetry constraints can be very helpful. By analyzing different examples with the toolbox you can derive strong conjectures about whether some theoretical restrictions hold.

Ideally, the toolbox would allow you to study repeated games with the scientific method of empirical falsification approach to game theory: as long as you don't find a counter-example against your hypothesis you can put a stronger belief that its true. I must say only "ideally", since approximation and discretization errors (or programming errors) may sometimes seem to falsify a hypothesis even if it holds in the continuous version of the game.[8] So one needs some experience and game theoretic intuition to correctly interpret the results. And in the end... as game theorists, we finally should go proving our conjectures!

Since the stage game is symmetric there are two natural conjectures:

1. For the equilibrium state, we can restrict attention w.l.o.g. to symmetric action profiles as candidates for optimal profiles.

2. For the punishment state of player i, we can restrict attention w.l.o.g. to action profiles where all punishing players $j \neq i$ choose the same output.

There are two ways to use the toolbox get insights into these hypothesis. First, we can look whether the optimal action profiles that we found by solving the unrestricted problem satisfy the postulated restrictions. If this holds, I would strengthen my belief that we can impose those restrictions w.l.o.g. If it does not hold, it does not necessarily prove that the conjectures are wrong, since there may be multiple optimal action profiles and the algorithm may have selected some that do not satisfy the condition.

The second way is to solve a restricted model that imposes the postulated restriction and to check whether the resulting payoff sets are the same than for the unrestricted problem. If the implementable payoff sets are smaller in the restricted problem (to a degree that seems not likely to be explained by approximation errors), it suggests that the conjecture was wrong.[9]

[8] Approximation and discretization errors, will be more of an issue for large stage games with imperfect public monitoring (see, e.g. Section 8) since computation time and memory restrictions but much stronger limits on the maximal grid sizes. I have no clue, how to develop a local grid refinement method for general games with imperfect monitoring. The problem is that alternative action profiles appear in the action constraints of the linear programs that calculate liquidity requirements and implementable payoffs. For games with perfect monitoring that problem can be circumvented, since our theoretical paper has shown that liquidity requirements only depend on stage game payoffs and stage game best-reply payoffs.

[9] Alternatively, a programming error took place. Like a mathematical proof, you should always check, whether there is a mistake somewhere in the code. Like in a mathematical

Let us start with the first method. The following code, plots (for the previously solved model `m.grid`) the outputs of players 2 and 3 against the output of player 1 in the equilibrium state and player 1's punishment state:

```
# Equilibrium state
plot(m.grid,xvar="e.q1",yvar=c("e.q2","e.q3"),col=c("blue","red"),
     main="Opt. equilibrium profiles")
abline(a=0,b=1)
# Punishment state
plot(m.grid,xvar="1.q1",yvar=c("1.q2","1.q3"),col=c("blue","red"),
     main="Opt. punishment profiles", legend.pos="topright")
```

The resulting plots are shown in Figure 8.

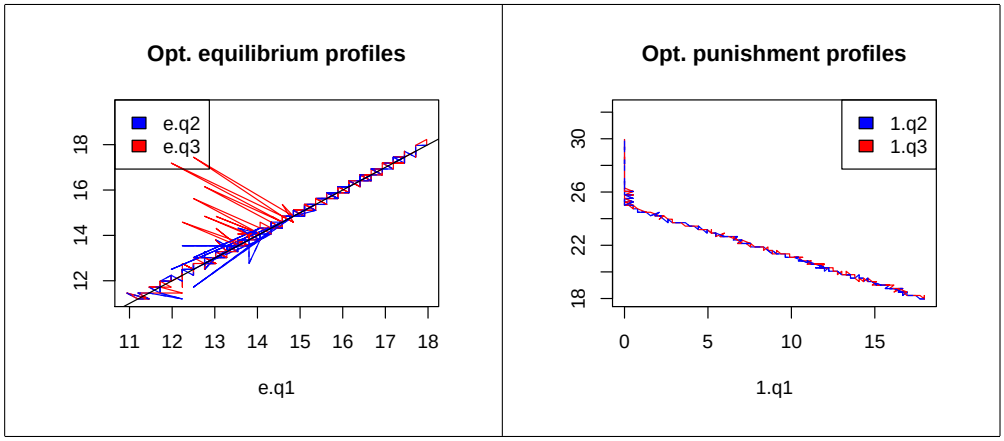


Figure 8: Plotting firms' optimal outputs in equilibrium state and punishment state against each other

The plot on the left hand side suggests that perhaps w.l.o.g., we could restrict attention to symmetric equilibrium action profiles. The command `abline(a=0,b=1)` simply did draw the black diagonal line with slope 1 in the left hand side. You can check whether these strange spikes begin to vanish, when you evaluate a finer grid. Similarly, the right hand side plot indicates that in firm i's punishment state it is optimal that the punishing firms chose the same output, while the punished firm chooses a different output.

Let us now solve a restricted model that imposes these symmetry constraints. Below you find the essential parts of the code, that you have to combine yourself with the code from the previous function to make it run.

proof, you may not always find every mistake however. Helpful for the search for bugs is that the program may throw an error or delivers strange outputs. Actually, debugging in R is quite fun, and I typically learn most about a problem after I have debugged my code. I give some hints on debugging in Appendix A.

```
pmx.init.cournot.sym = function(n=2, q.range, A, B, MC) {
  # First part

  #... insert code from first part of pmx.init.cournot

  # Second part: Code that specifies symmetry constraints

  # In equilibrium state only player 1 chooses freely his
  # action other firms are assumed to choose symmetric actions
  x.free.e = 1
  link.fun.e = function(xf.mat,k=NULL) {
    # xf.mat will only have one column, which
    # contains the actions of firm 1
    # return a matrix with n cols that are all identical to
xf.mat
    return(matrix(as.vector(xf.mat),NROW(xf.mat),n))
  }

  # In the punishment states only the punished player and one
other player
  # (here either player 1 or 2) can freely decide on their
actions
  x.free.i = lapply(1:n, function(i) c(1,i))
  x.free.i[[1]] = c(1,2)
  link.fun.i = function(xf.mat,k) {
    if (k == 1) {
      mat = matrix(as.vector(xf.mat[,2]),NROW(xf.mat),n)
      mat[,1] = xf.mat[,1]
    } else {
      mat = matrix(as.vector(xf.mat[,1]),NROW(xf.mat),n)
      mat[,k] = xf.mat[,k]
    }
    return(mat)
  }

  m = pmx.init.game(n=n,nx=n,gx.fun=gx.fun,cx.fun=cx.fun,
              x.range=q.range, symmetric=TRUE,
              x.free.e = x.free.e, link.fun.e=link.fun.e,
              x.free.i = x.free.i,link.fun.i = link.fun.i,
              name=paste("Cournot Game (",n," firms)",sep=""),
              names.x=paste("q",1:n,sep=""))
  return(m)
}
# Parameters of the model
n=100; A=100; B=1; MC=10;
m = pmx.init.cournot(n=n,q.range=c(0,A / (n-1)),A=A,B=B,MC=MC)
```

The basic philosophy is that you can specify for each state a set of free actions and a link function that specifies the remaining actions of an action profile as function of the free actions. Under restriction to symmetric action profiles on the equilibrium state, we only need one free action profile. The line `x.free.e=1` says that player 1's action shall be free. The function `link.fun.e` must be a function that takes (besides the state k) as argument a matrix `xf.mat` that specifies the free actions. Since only one action is free, the matrix will only have one column and each row will correspond to a different action of player 1 (the function will be repeatedly called with different actions as the algorithm unfolds). The function must return a matrix that specifies for each row of xf.mat the complete action profile. Here, simply every player chooses the same action profile than player 1.

For the punishment state of player i we specify to free actions: the action of player i and some other player (the other player is 1 unless $i = 1$). The link function `link.fun.i` returns a matrix in which all punishing players choose the same outputs.

If you run the model, you can indeed verify that the resulting optimal payoffs are the same than for the unrestricted model. To compare the models, store the solved unrestricted and restricted models in variables `m.u` and `m.r`. One way to compare the payoffs, is to tell R to record the plotting history, then to plot the two models after each other and compare the graphs by switching between the plots. (There is also a more convenient function `plot.compare.models`, but I have not yet implemented it for perfect monitoring games).

The restricted model can be easily solved for a much larger number of firms, since the number of free actions remains constant. Try solving it for $n = 100$. The only slight annoyance you might encounter is that the option `use.random.start.points = TRUE` leads to slight inaccuracies in the calculation of punishment profiles if the number of firms is large.

Exercise 5.1. Check different specifications of the parameters of `cnt` when solving the symmetric model for a large number of firms and find a robust one that is considerably quick. Also try out the method `refine.solution` for refining a solution with different solution methods. Perform some graphical comparative statics of the payoff sets with respect to the number of firms and other parameters that might interest you.

In Section 8, we will compare repeated Cournot oligopolies with perfect and imperfect monitoring. Some more exercises will be given there.

6 Hotelling Competition and Multi Market Contact

This Section studies collusion in one or several markets that each are described by a Hotelling model. It exemplifies how one can model stage games where

players can have multi-dimensional activity spaces. This Section also illustrates methods that help to check whether one correctly specified the cheating payoffs of the stage game.

6.1 The Hotelling Stage Game

Consider 2 (or more) firms that compete with each other in 1, 2 or more different markets. In each market only two firms are active. Demand in each market is independent from prices in other markets and described by a linear Hotelling model.

More specifically, let us assume that in any given market, consumers are uniformly distributed over the unit interval $[0, 1]$ and that the two active firms are located at the end points of the interval. For the following illustration, assume firm 1 is located at position 0 and firm 2 at position 1. A consumer at position $x \in [0, 1]$ gets a net benefit of $w - p_1 - x\tau$ and $w - p_2 - (1 - x)\tau$ when buying a good from firm 1 or 2 at prices p_1 and p_2, respectively. The parameter w measures the consumers' gross valuation for a good and τ is a measure of transportation costs. Each consumer has unit demand and buys at most from one firm and the mass of total consumers shall be given by a market size parameter S.

In an interior equilibrium, where both firms sell positive amounts and every consumer buys a product, firm i's demand is given by

$$q_i^c(p_i, p_j) = \left(\frac{1}{2} + \frac{p_j - p_i}{2\tau}\right) S$$

If consumers' valuation w is too low, compared to the competitor's price, the monopolistic demand

$$q^m(p_i) = \left(\frac{w - p_i}{\tau}\right) S$$

becomes instead relevant for firm i. We also have to account for the boundary conditions that demand cannot be negative and cannot exceed S. We thus find that firm i's demand in the market is given by

$$q_i(p_i, p_j) = \max\left\{\min\left\{q_i^c(p_i, p_j), q_i^m(p_i), 1\right\}, 0\right\} S.$$

Firm i has constant marginal costs of production c_i. We assume that prices are restricted to $0 \leq p_i \leq w$.

6.2 Finding errors in the best-reply functions

When Hotelling models are studied in one-shot games, it is usually assumed that the customers' gross value w is relatively large compared to the Nash equilibrium prices and that both firms have identical cost. In that case, the relevant best-reply functions can be found under the assumptions that all consumers buy from one of the two firms and that there is an interior consumer $\hat{x} \in (0, 1)$ who is indifferent between buying from either of the two firms. Then

using the first order conditions, one finds that firm i's best-reply price is given by

$$\hat{p}_i^c(p_j) = \frac{1}{2}(p_j + c_i + \tau)$$

If we want to study repeated games the actual value of w becomes relevant for collusive prices and we may have to account for the fact that best-reply functions have a more complicated form. Nevertheless, to illustrate some debugging facilities, let us believe for the moment that the function above indeed specifies the best reply functions of the stage game. Then the code below initializes a Hotelling model with a single market and solves the repeated game.

```
init.hotelling = function(ms,symmetric=FALSE) {
  colnames(ms) = c("market","firm1","firm2",
                   "MC1","MC2","tau","size","w")
  n = max(ms[,c("firm1","firm2")])

  store.objects()
  #restore.objects("init.hotelling")

  # Stage game payoffs
  gx.fun = function(p) {
    store.objects("gx.fun")
    #restore.objects("gx.fun")
    g.mat = matrix(0,NROW(p),n)
    for (ma in 1:NROW(ms)) {
      tau = ms[ma,"tau"];size = ms[ma,"size"]; w = ms[ma,"w"];
      p1 = p[,ma*2-1];p2 = p[,ma*2];
      q1 = pmax(0,pmin((w-p1)/tau,
                pmin(1,(1/2+(p2-p1)/(2*tau)))))*size
      q2 = pmax(0,pmin((w-p2)/tau,
                  pmin(1,(1/2+(p1-p2)/(2*tau)))))*size
      g.mat[,1] = g.mat[,1]+q1*(p1-ms[ma,"MC1"])
      g.mat[,2] = g.mat[,2]+q2*(p2-ms[ma,"MC2"])
    }
    g.mat
  }

  # Stage game cheating payoffs.
  #Assuming FOC specfies best reply prices
  cx.fun = function(p) {
    store.objects("cx.fun")
    #restore.objects("cx.fun");
    #restore.objects("init.hotelling")
    c.mat = matrix(0,NROW(p),n)
    ma = 1
```

```
    while(ma <= NROW(ms)) {
      tau = ms[ma,"tau"];size = ms[ma,"size"];w = ms[ma,"w"];
      MC1 = ms[ma,"MC1"]; MC2 = ms[ma,"MC2"];
      p1 = p[,ma*2-1];p2 = p[,ma*2];
      p1.br = (1/2)*(p2+MC1+tau)
      p2.br = (1/2)*(p1+MC2+tau)
      q1 = pmax(0,pmin((w-p1.br)/tau,
                pmin(1,(1/2+(p2-p1.br)/(2*tau)))))*size
      q2 = pmax(0,pmin((w-p2.br)/tau,
                pmin(1,(1/2+(p1-p2.br)/(2*tau)))))*size

      c.mat[,1] = c.mat[,1]+q1*(p1.br-MC1)
      c.mat[,2] = c.mat[,2]+q2*(p2.br-MC2)
      ma = ma+1
    }

    c.mat
  }
  name=paste("Hotelling (markets: ",NROW(ms),")", sep="")
  nx = NROW(ms)*2
  name.grid = make.grid.matrix(x=list(1:NROW(ms),1:2))
  names.x = paste("m",name.grid[,1],".p",name.grid[,2],sep="")
  x.range = cbind(0,rep(ms[,"w"],each=2))
  m =
pmx.init.game(n=n,nx=nx,x.range=x.range,symmetric=symmetric,
        gx.fun=gx.fun,cx.fun=cx.fun,
        name=name, names.x=names.x)
  return(m)
}
ms = matrix(c(
 #market  firm1, firm2,   MC1,  MC2, tau, size,  w,
       1,     1,     2,      10,  10, 100, 100,   200
     ),ncol=8,byrow=TRUE)
m = init.hotelling(ms=ms,symmetric=TRUE)
cnt = list(method="grid", step.size.start=10,
           step.size.end = 0.5, num.refinements = 1)
m = solve.game(m,cnt=cnt)
plot(m)
m.unsure = m
```

Each row in the matrix `ms` specifies the structure of one particular market. Here, we specified that there is only one market.

The function `solve.game` runs on my computer without that the toolbox throws any error and I would say that the plot of equilibrium payoffs looks reasonable... Does that mean, we specified the correct best-reply function? Not necessarily. The following code allows to search for errors in the best

reply-function:

```
mat = check.cx.fun(m,num=30,method="grid",num.grid.steps=10000)
```

The function `check.cx.fun` randomly samples `num` action profiles and calculates for those action profiles the cheating payoffs for randomly selected players applying a numerical optimization procedure using the specified payoff function `m$gx.fun` (by default it performs a grid search over the actions of the selected player). Then it compares these numerical results with the cheating payoffs specified by `m$cx.fun`. Information about the selected points and cheating payoffs is returned in a matrix. For example, type

```
round(mat[,"diff"],3)
```

```
##  [1]     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
##  [9]     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
## [17]     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
## [25]     0.0     0.0  -361.6 -2255.2 -2330.2 -2277.7
```

to see the vector of differences of the cheating payoffs specified by `cx.fun` and the cheating payoffs from the grid search. The function `check.cx.fun` also creates a plot that allows you compare the differences in calculated cheating payoffs graphically.

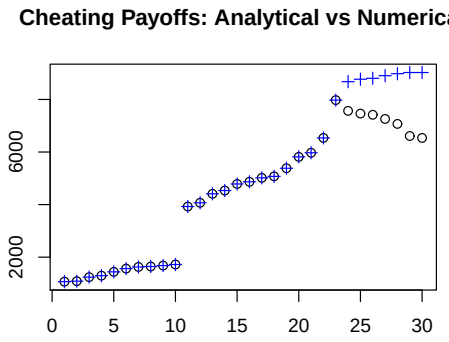


Figure 9: Checking specified cheating payoff functions

The black circles are the analytical cheating payoffs and the blue crosses the corresponding numerical results. If the blue crosses lie on the black circles (or slightly below, to a degree that depends on the coarseness of your grid search) everything is fine. If this is not the case everywhere (like on the right hand side of the shown plot), you misspecified the cheating payoffs.

Does this result mean that the calculated set of equilibrium payoffs is incorrect? Not necessarily, the plot indicates that at least for some subset of action profiles, we specified cheating payoffs correctly. By calling

```
mat = check.cx.fun(m,x.sample="opt.mat",
                   method="grid",num.grid.steps=10000)
round(mat[,"diff"],5)
```

you can verify that the cheating payoffs were correctly calculated for the optimal action profiles. That means that there indeed exist equilibria that can implement the calculated optimal payoffs. However, we might have found a larger payoff set, if we would have specified a cheating payoff function that is correct for all action profiles.

6.3 Specifying cheating payoff functions that use numerical optimization

You might wonder, why you have to specify a closed-form solutions for cheating payoffs if cheating payoffs can also be calculated by a grid search or some other numerical optimization procedure.... Actually, it is possible to specify a function that calculates cheating payoffs by numerical optimization or grid search. The drawbacks are that solving the game can take considerably more time and there is no guarantee that the resulting payoff sets are inner approximations to the payoff sets of the continuous game if you only use a coarse grid search. The lines

```
m.num = init.hotelling(ms=ms,symmetric=TRUE)
m.num$cx.fun = make.numerical.cx.fun(m.num,num.grid.steps=10000)
```

tell the model to use a `cx.fun` that calculates cheating payoffs using a grid search over `gx.fun`. So far, the function only works using a grid search and I only implemented it for games where each player has a single activity.

You will find that it takes considerably longer to solve the repeated game than for the case that closed-form solution for stage game cheating payoffs are provided. If you have two player games and don't know the cheating payoffs, you may rather prefer to use the methods explained in Sections 3, 4 and 5.1 to solve the game.

6.4 Specifying correct best-reply functions

Let us now specify a correct closed-form function for stage game cheating payoffs. For simplicity, let us allow firms to set only prices that satisfy $p_1, p_2 \leq w$. Then given a price p_j of the competitor, firm i has several candidates for a best-reply:

1. Set the monopoly price $p_i^m = \frac{w+c_i}{2}$. This is optimal when given prices p_j and firm i's monopoly price, not the whole market would be covered, i.e. if $q^m(p_i^m) + q^m(p_j) \leq S$. This condition is equivalent to

$$p_j \geq \frac{3}{2}w - \tau - \frac{1}{2}c_i$$

2. Chose that price that gives firm i exactly a market share of $S - q^m(p_j)$. This price is given by
$$\hat{p}_i^k = 2w - \tau - p_j$$
Note that firm i's demand function has a kink at $\hat{p}_i^k$.

3. Choose the interior best-reply price, which we already specified above
$$\hat{p}_i^c(p_j) = \frac{1}{2}(p_j + c_i + \tau)$$

4. Choose the highest price with which firm i can steal the whole market from firm j:
$$\hat{p}_i^s = p_j - \tau$$

The following modified cheating payoff function accounts for all four cases:

```
# Stage game cheating payoffs. Assuming FOC specifies best reply
prices
  cx.fun = function(p) {
    store.objects("cx.fun")
    #restore.objects("cx.fun");
    #restore.objects("init.multimarket.hotelling")
    c.mat = matrix(0,NROW(p),n)
    ma = 1
    while(ma <= NROW(ms)) {
      tau = ms[ma,"tau"];size = ms[ma,"size"];w = ms[ma,"w"];

      for (i in 1:2) {
        j = 3-i
        # Not very elegant but who cares...
        if (i == 1) {
          MC.i = ms[ma,"MC1"]; MC.j = ms[ma,"MC2"];
          p.i = p[,ma*2-1];p.j = p[,ma*2];
        } else {
          MC.j = ms[ma,"MC1"]; MC.i = ms[ma,"MC2"];
          p.j = p[,ma*2-1];p.i = p[,ma*2];
        }

        # Try out the different best replies

        # kink
        p.i.br = 2*w-tau-p.j
        qi = pmax(0,pmin((w-p.i.br)/tau,
                  pmin(1,(1/2+(p.j-p.i.br)/(2*tau)))))*size
        pi.k = qi*(p.i.br-MC.i)

        # interior
```

```
        p.i.br = ((tau+MC.i+p.j)/2)
        qi = pmax(0,pmin((w-p.i.br)/tau,
                  pmin(1,(1/2+(p.j-p.i.br)/(2*tau)))))*size
        pi.c = qi*(p.i.br-MC.i)

        # steal the whole market from the other player
        p.i.br = p.j-tau
        qi = pmax(0,pmin((w-p.i.br)/tau,
                    pmin(1,(1/2+(p.j-p.i.br)/(2*tau)))))*size
        pi.s = qi*(p.i.br-MC.i)

        # Set monopoly price in those rows where pj is very high
        pi.m = rep(0,NROW(p))
        rows = (p.j >= (3/2)*w-tau-(1/2)*MC.i)
        p.i.br = (w+MC.i)/2
        qi = pmax(0,pmin((w-p.i.br)/tau,
                  pmin(1,(1/2+(p.j-p.i.br)/(2*tau)))))*size
        pi.m[rows] = (qi*(p.i.br-MC.i))[rows]

        # Take that best-reply that yields highest profits
        c.mat[,i] =
c.mat[,i]+pmax(pi.k,pmax(pi.c,pmax(pi.s,pi.m)))
      }
      ma = ma +1
    }
    return(c.mat)
  }
```

Exercise 6.1. Include the corrected function from above in the function `init.hotelling`. Check whether you can find any mistake in the cheating payoff function by using `check.cx.fun`. Using the function `plot.compare.model`, compare the payoff sets using the wrong specification of cheating payoffs, the cheating payoff function generated by `make.numerical.cx.fun` and the correct specification of cheating payoffs (try the comparison for different values of w, for low values of w you should not be able to solve the model with the wrongly specified cheating payoffs). Using the correctly specified cheating payoffs, perform comparative statics of the payoff set w.r.t. transportation cost (are maximal payoffs monotone in τ?) and other variables of the model.

6.5 Multi-market Collusion

The code below solves two models that differ by their transportation costs τ:

```
cnt = list(method="grid", step.size.start=10,step.size.end =
0.5)
```

```
ms = matrix(c(
 #market  firm1, firm2,   MC1,  MC2, tau, size, w,
       1,     1,      2,       10,  10, 100, 100,   400
     ),ncol=8,byrow=TRUE)
m = init.hotelling(ms=ms,symmetric=TRUE)
m.high = solve.game(m,cnt=cnt)
ms = matrix(c(
 #market  firm1, firm2,   MC1,  MC2, tau, size, w,
       1,     1,      2,       10,  10, 10,  100,   400
     ),ncol=8,byrow=TRUE)
m = init.hotelling(ms=ms,symmetric=TRUE)
m.low = solve.game(m,cnt=cnt)
plot.compare.models(m.high,m.low,xvar="delta",yvar="Ue",
                    m1.name="high tau",m2.name="low tau",
                    legend.pos = "bottomright")
```

The resulting payoff comparison is shown in Figure 10:

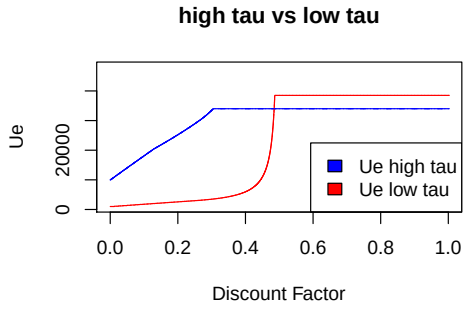


Figure 10: Maximal Collusive Payoffs under high vs low transportation costs

High transportation costs are preferable for firms under low discount factors as they facilitate high prices. However, as the discount factor grows large, firms' profits are higher under lower transportation costs since more surplus can then be extracted from consumers.

The code below solves a model where the same pair of firms is active in both market and can simultaneously collude in both markets.

```
# Parameters of the model
ms = matrix(c(
 #market  firm1, firm2,   MC1, MC2, tau, size, w,
       1,     1,      2,      10,  10, 100, 100,   400,
       1,     1,      2,      10,  10,  10, 100,   400
     ),ncol=8,byrow=TRUE)
```

```
m = init.hotelling(ms=ms,symmetric=TRUE)
cnt = list(method="grid", step.size.start=5,step.size.end = 2,
           num.refinements = 1,use.random.start.points = TRUE,
           num.random.start.points=10000)
m = solve.game(m,cnt=cnt)
m.mult = m
plot(m.mult)
```

We solved the model using a grid search. You can always try to refine a solution, e.g. by using a different solution methods or different parameters. The following code tries out whether the a few random samples find better action profiles:

```
# As a check, try to refine the solution with random sampling
cnt = list(method="random", step.size=0.1,
           size.draw = 100,num.draws = 50,
           local.abs.width=4, prob.draw.global = 1)
m.mult = refine.solution(m.mult,cnt)
```

Most likely you won't see any relevant improvement in the graphical representation of the solution. The grid search seems to have worked fine. The following lines test our specification of the cheating payoffs:

```
mat = check.cx.fun(m.mult,num=30,
                   i.of.x = c(1,2,1,2), use.i = c(1,2),
                   method="grid",num.grid.steps=300)
```

You can repeat the function call above several times. The resulting plots should indicate that everything is fine.

To see whether multi-market contact facilitates collusion, it would be nice to compare the sum of payoffs from single market collusion with the maximal payoffs in the multi-market collusion model. The code below generates such a plot (shown in Figure 11):

```
# Generate matrices for same delta.sequence
delta = seq(0,1,by=0.01)
mat1 = get.mat.of.delta(m.high,delta =
delta,add.crit.delta=FALSE)
mat2 = get.mat.of.delta(m.low,delta =
delta,add.crit.delta=FALSE)
mat.b = get.mat.of.delta(m.mult,delta =
delta,add.crit.delta=FALSE)
mat.s  = mat1
mat.s[,"Ue"]= mat1[,"Ue"]+mat2[,"Ue"]
# Draw the different matrices
ylim = range(c(mat.s[,"Ue"],mat.b[,"Ue"]))
```

```
plot(mat.b[,"delta"],mat.b[,"Ue"],col="blue",type="l",
     main="Single Market vs Multi Market Collusion",
     xlab="delta",ylab="Ue",ylim=ylim)
lines(mat.s[,"delta"],mat.s[,"Ue"],col="red",lty=1)
lines(mat.b[,"delta"],mat.b[,"Ue"],col="blue",lty=2)
lines(mat1[,"delta"],mat1[,"Ue"],col="green",lty=2)
lines(mat2[,"delta"],mat2[,"Ue"],col="lightgreen",lty=2)
```

We find that even though we already have side payments, multi-market contact can still increase collusive profits. This is the case for those discount factors for which in the single market with high transportation costs perfect collusion can be achieved but not in the single market with low transportation costs.

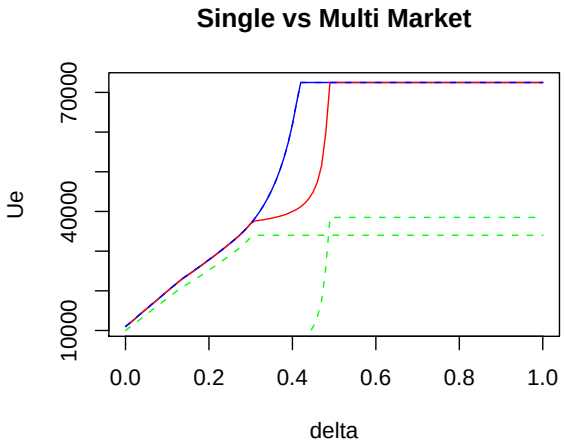


Figure 11: Added collusive profits in the two separate markets vs profits under multi-market collusion

Part II
Imperfect Public Monitoring

This part illustrates how the toolbox can be used to solve infinitely repeated games with imperfect public monitoring and monetary transfers. So far, I have only implemented our algorithm that characterize the payoff set for the case that money burning is possible, but not yet the algorithm described in Section 6 of Goldluecke & Kranz (2010) that deals with the case of no money burning.

7 A Noisy Prisoners' Dilemma Game

To illustrate how the package repgame can solve repeated games with imperfect public monitoring, let us first consider the noisy prisoners' dilemma game that has been analytically solved in Goldluecke & Kranz (2010). There are two players who can choose in each period to either cooperate (C) or defect (D). Expected payoffs in the stage game are given by the following normalized payoff matrix:

	C	D
C	1,1	-s,1+d
D	1+d,-s	0,0

with $d, s > 0$ and $d - s < 1$. Players do not publicly observe the played action profile, but only a signal y that can take 4 different values: y_C,y_D,y_1 and y_2. The probability distribution of the signal $\phi(y|a)$ depends on the played action profile and is described in the following table

$\phi(y\|a)$	(C,C)	(C,D)	(D,C)	(D,D)
y_C	$1-\lambda-2\alpha$	$1-\lambda-\mu-2\alpha-\beta$	$1-\lambda-\mu-2\alpha-\beta$	$1-\psi$
y_D	λ	$\lambda+\mu$	$\lambda+\mu$	ψ
y_1	α	α	$\alpha+\beta$	0
y_2	α	$\alpha+\beta$	α	0

with $0 < \lambda \leq \lambda+\mu$ and $0 < \alpha \leq \alpha+\beta$ and $1-\lambda-\mu-2\alpha-\beta \geq 0$. To interpret the signal structure, assume mutual cooperation (C,C) shall be implemented in the static problem. The signal y_D is an anonymous indicator for defection: y_D becomes more likely if some player unilaterally defects but its probability distribution does not depend on the identity of the deviator. The parameter λ can be interpreted as the probability of a type-one error, i.e. the probability that y_D is observed even if no player defected. The parameter μ measures by how much the likelihood of y_D increases if some player unilaterally deviates. The signal y_i is an indicator for unilateral defection by player i. Like λ, the parameter α can be interpreted as the probability of a type-one error, i.e. the probability to wrongly get a signal for unilateral defection of player i. Similar to μ, the parameter β measures by how much the likelihood of y_i increases if player i unilaterally deviates from mutual cooperation.

The following function initializes such a Noisy PD game:

```
init.noisy.pd.game = function(d,s,lambda,mu,alpha,beta,psi) {
  store.objects("init.noisy.pd.game")
  # restore.objects("init.noisy.pd.game")

  # Payoff matrix of player 1 (that of player 2 is symmetric)
  g1 = matrix(c(
    1,    -s,
    1+d,   0),2,2,byrow=TRUE)
  lab.ai = c("C","D")


  # Create phi.mat, which stores the signal distributions
  prob.yC = c(1-2*alpha-lambda, 1-2*alpha-lambda-mu-beta,
              1-2*alpha-lambda-mu-beta, 1-psi)

  phi.mat = matrix(c(
    #CC       CD          DC          DD
    prob.yC,                                  #yC
    lambda,   lambda+mu,  lambda+mu , psi,    #yD
    alpha,    alpha,      alpha+beta, 0,      #y1
    alpha,    alpha+beta, alpha     , 0       #y2
  ), 4,4,byrow=TRUE)

  lab.y = c("yC","yD","y1","y2")

  m=init.game(g1=g1,phi.mat=phi.mat, symmetric=TRUE,
              lab.ai = lab.ai,lab.y=lab.y, name="Noisy PD")
  m
}
```

The matrix `phi.mat` has as many rows as signals and as many columns as action profiles. So `phi.mat[y,a]` denotes the probability that signal `y` is realized if action profile `a` is played. The following code initializes one noisy PD game, solves the model and draws two different plots of the solution.

```
d=1;s=1.5;lambda=0.1;mu=0.2;alpha=0.1;beta=0.2;psi = 1
m = init.noisy.pd.game(d,s,lambda,mu,alpha,beta,psi)
m = solve.game(m)
```

```
## [1] "solve.lp  get.L.and.B get.L(B=0) a=C|C"
## [1] "solve.lp  get.L.and.B: get.L a=C|C"
## [1] "solve.lp  get.L.and.B get.B(L.min) a=C|C L.min = 5"
## [1] "solve.lp  get.L.and.B get.L(B=0) a=C|D"
## [1] "solve.lp  get.L.and.B: get.L a=C|D"
## [1] "solve.lp  get.L.and.B get.L(B=0) a=D|C"
```

```
## [1] "solve.lp  get.L.and.B: get.L a=D|C"
```

```
plot(m,xvar="L",yvar=c("Ue","V"),lwd=2)
plot(m,xvar="delta",delta.seq=seq(0.5,1,by=0.001),
     yvar=c("Ue","V"),lwd=2)
```

The first plot shows the joint equilibrium and punishment payoffs as function of the totally available liquidity L, the second plot as function of the discount factor δ (see Figure 12).

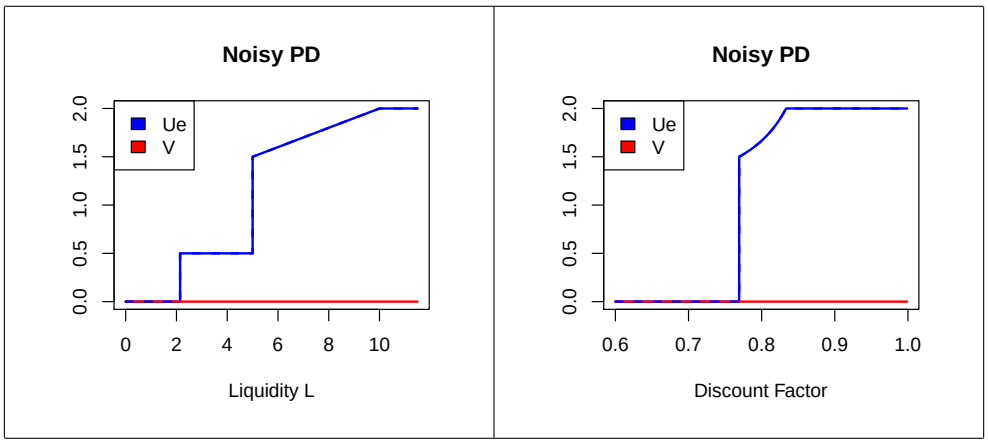


Figure 12: Payoffs of the repeated Noisy PD Game

For readers who are familiar with our theoretical paper, let me explain more on the structure of optimal equilibria. Information about optimal action plans is stored in `m$opt.mat`. Let us look at some of its columns:

```
m$opt.mat[,c("L","delta","Ue","V","v1","v2","opt","helper")]
```

```
##                    L  delta  Ue V v1 v2 opt helper
## (D|D),(D|D)    0.000 0.0000 0.0 0  0  0   1      0
## ---            2.143 1.0000 0.0 0  0  0   0      1
## (C|D),(D|D)    2.143 0.8108 0.5 0  0  0   0      0
## ---            5.000 0.9091 0.5 0  0  0   0      1
## (C|C),(D|D)    5.000 0.7692 1.5 0  0  0   1      0
## (C|C),(D|D)   10.000 0.8333 2.0 0  0  0   1      0
```

The row names indicate the optimal action plan. Since the game is symmetric, only the optimal equilibrium action profile a^e and the punishment profile for player 1 a^1 are shown. A row with name "—" corresponds to a helper point that describes the end of a horizontal line in Figure 12 (left). This means that the action plan remains optimal in the corresponding liquidity range and joint equilibrium payoffs and punishment payoffs remain constant (i.e. there

is no money burning). After row 5 there is no helper point, which indicates that there is a segment between L=5 and L=10, in which U^e linearly increases in L. Payoffs increase because less money burning is required in the static problem if more joint liquidity becomes available. In the optimal stationary equilibrium corresponding to row 5, mutual cooperation (C, C) will be played on the equilibrium path, but in expectation $2 - 1.5 = 0.5$ units of money will be burned every round. Row 6 corresponds to an optimal equilibrium with the same action plan but no money burning.

The column `opt` indicates whether the action plan that is optimal for the given liquidity L is indeed an optimal action plan for some discount factor δ. In our example, row 3 with $a^e = (C, D)$ is not optimal for any discount factor, since it has a higher critical discount factor $\delta^* = 0.81$ than row 6 with $\delta = 0.77$. Thus, if you look at the right hand side plot of Figure 12, you will find that the joint payoff of 0.5 associated with the third row does not appear.

Exercise 7.1. Perform some graphical comparative statics with respect to the different parameters of the model (see Section 4.2) and compare the optimal equilibria with the analytical closed-form solutions in Goldluecke & Kranz (2010).

8 Collusion without observing sales

The classical paper by Green and Porter (1984) used repeated games to model collusion between quantity setting firms that cannot observe other firms sales but only publicly observe the realized market prices. The realized market price is only a noisy signal of total output, since it also depends on an unobservable demand shock. Optimal equilibria in this set-up have already been studied by Abreu, Pearce and Stachhetti (1986), who exploit the fact that price signals are completely uninformative with respect to which firm deviated.

While the analysis in this Section may therefore not yield many new economic insights, I think this class of models is well suited to illustrate how to use my toolbox for analyzing discrete approximations of games with imperfect monitoring and continuous action spaces.

8.1 Simple initial example

In the following exercise you can first explore a very simple variant of a model of collusion without observing sales.

Exercise 8.1. Consider two firms that can in each period either choose low, medium or high output levels. Expected payoffs are as in Section 3 given by:

		Firm 2		
		q_L	q_M	q_H
Firm 1	q_L	**10,10**	**3,15**	**0,7**
	q_M	**15,3**	**7,7**	**-4,5**
	q_H	**7,0**	**5,-4**	**-15,-15**

Firms cannot observe other firms' past output level, but only observe a publicly realized market price. Assume the realized price can take only two levels $p \in \{p_L, p_H\}$. Assume $q_L = 1$, $q_M = 2$, and $q_H = 3$ and let the probability of a low price be given by

$$Pr(p = p_L|(q_1, q_2)) = (q_1 + q_2)\Delta$$

where Δ is a parameter with $0 < \Delta < 6\Delta < 1$. Write a function that initializes the model. Solve the model and investigate the comparative statics w.r.t. Δ. Using the results in our theoretical paper, try to find analytical solutions for $L(a)$, $U^e(L|a)$ and $v_i(L|a)$. Compare them with the results from the computer program.

8.2 Initializing duopolies with larger action and signal spaces

We now illustrate the analysis of repeated games with larger action and signal spaces. Consider 2 symmetric firms that produce a homogeneous good. Production costs are given by a convex, increasing function $c(q_i)$ and each firm has the same capacity limit q_{max}. The market demand function shall be given by

$$D(P) \quad = \quad e^s D_0(P).$$

$D_0(P)$ is a deterministic component of the demand function and s is a random variable that measures the market size and follows some distribution with c.d.f. $F_s(s)$. We assume that $D(P)$ is strictly decreasing. The inverse demand function, which depends on the totally produced quantity $Q = q_1 + q_2$ and the realized market size s, is then given by

$$P(Q, s) = D_0^{-1}\left(e^{-s}Q\right).$$

In the action stage, firms simultaneously choose quantities they bring to the market without knowing the realization of the total market demand s. The distribution of market prices given total production Q is given by

$$\begin{aligned} F_p(p|Q) &= Pr\left\{P(Q, s) \leq p\right\} \\ &= Pr\left\{e^{-s}Q \leq D_0(p)\right\} \\ &= F_s\left(log\left(\frac{Q}{D_0(p)}\right)\right) \end{aligned}$$

If we do not allow for negative demand and negative market prices, the formula gets adapted as follows:

$$F_p(p|Q) = \begin{cases} 1 & \text{if } p \geq P_{max} \\ F_s\left(log\left(\frac{Q}{D_0(p)}\right)\right) & \text{if } 0 \leq p \leq P_{max} \\ 0 & \text{if } p < 0 \end{cases} \tag{3}$$

where P_{max} is the choke price at which demand gets zero, i.e. $D_0(P_{max}) = 0$.

Example. For example assume the deterministic component of the demand function is linear:

$$D_0(P) \quad = \quad max\left\{0, \alpha - \beta P\right\}$$

The choke price is then given by $P_{max} = \frac{\alpha}{\beta}$ and the distribution of market prices is given by formula (3).

The toolbox cannot directly solve repeated games that have continuous action or signal space. We therefore have to discretize both. To discretize each firm's action space, we can simply choose some finite grid of quantities between 0 and q_{max}. There are different methods to discretized a continuous signal distribution.

The toolbox contains a function that performs discretization according to the following method. First, we specify a finite grid of M price signals $\{y_1, ..., y_M\}$ with $0 = y_1 < ... < y_M$. If a choke price P_{max} exists, we set $y_M = P_{max}$. The probability weight on each price signal y_m as function of total produced output Q shall be given by

$$\phi(y_m|Q) \quad = \quad \begin{cases} F_p(y_m) + \frac{F_p(y_{m+1}) - F_p(y_m)}{2} & \text{if } m = 1 \\ \frac{F_p(y_m) - F_p(y_{m-1})}{2} + \frac{F_p(y_{m+1}) - F_p(y_m)}{2} & \text{if } 1 < m < M \\ \frac{F_p(y_m) - F_p(y_{m-1})}{2} + 1 - F_p(y_m) & \text{if } m = M \end{cases}$$

Basically, the probability mass between two adjacent signals is divided equally between the two signals.

The following code initializes and solves a repeated game in this class.

```
# Require global parameters to be set
# D0.fun, p.max, F.s = F.s, cost.fun = cost.fun,q.max = q.max
# Either nq=nq or q.seq=NULL
init.payoffs.green.porter = function(make.q.seq=TRUE) {
  #store.objects()
  # restore.objects("init.payoffs.green.porter")

  # Generate the distribution function for prices F.p
  # The assignment <<- stores F.p in the global environment
  F.p <<- function(p,Q) {
    x = F.s(log(Q / D0.fun(p)))
    x[p >= p.max] == 0
    x
  }
  # Action space of each firm (we assume firms are symmetric)
  if (make.q.seq) {
    q.seq <<- seq(0,q.max,length=nq)
  }
```

```
  # Generate function that calculates expected payoffs for
  # each action profile
  # The function performs a very fine discretization of
  # the distribution F.p to approximate expected payoffs
  g.fun = function(qm) {
    store.objects("g.fun")
    #restore.objects("g.fun")

    Q = rowSums(qm)
    g = matrix(0,NROW(qm),2)

    # Calculate expected profits for every action profile
    # The loop takes some time, since calculation of an expected
price
    # uses numerical integration for every level of Q
    for (r in 1:NROW(qm)) {
       Ep = calc.mean.from.F.fun(F.p,Q=Q[r],x.min=0,x.max=p.max)
       g[r,] = Ep * qm[r,] - cost.fun(qm[r,])
    }
    # Manually correct the case that no firm produces anything
    if (Q[1]==0) {
      g[1,] = c(0,0)
    }
    g
  }

  m = init.game(g.fun=g.fun, action.val = q.seq, symmetric=TRUE,
        name="Green-Porter with Optimal Penal Codes")
  return(m)
}

init.signals.green.porter = function(m=m) {
  qm = m$action.val.mat
  Q=rowSums(qm)
  y.val <<- seq(0,p.max,length=ny)

  # Calculate the F.p at every signal y
  # mapply and apply work similar than Vecorize
  F.y = mapply(F.p,Q=Q,MoreArgs=list(p=y.val))

  # Assign probability weight according to the
  # discretization procedure explained in the tutorial
  phi.mat = apply(F.y,2,discretize.given.F.vec)

  # We have to manually correct the price distribution if
```

```
  # no firm produces anything
  phi.mat[,Q==0] = c(rep(0,ny-1),1)

  # Initialize the signal distribution for the model m
  m = set.phi.mat(m,phi.mat,lab.y = paste("y",y.val,sep=""))
  m$y.val = y.val
  return(m)
}

# Initialize parameters

# Deterministic part of demand function
alpha = 100; beta = 1;
D0.fun = function(P) {alpha-beta*P}
p.max = alpha / beta
q.max = alpha

# Distribution of market size
mean.s = 0; sigma.s = 0.2
F.s = function(x) {pnorm(x,mean.s,sigma.s)}

# Cost function
MC = 10
cost.fun = function(q) {MC*q}

# Number of actions per firm and number of signals
nq = 51;
ny = 21;

# Initialize model with perfect monitoring
m.pm = init.payoffs.green.porter()
m.pm$name = "Cournot with Payments & Optimal Penal Codes"

# Transform into a model with imperfect monitoring
m = init.signals.green.porter(m.pm)
m$name = "Green-Porter with Payments & Optimal Penal Codes"
```

Depending on your reference point, it may look like a lot of code, but I hope together with the comments it is quite intuitive what is going on.

8.3 Solving the model and analysing the solution

To solve the model above, we can simply type.

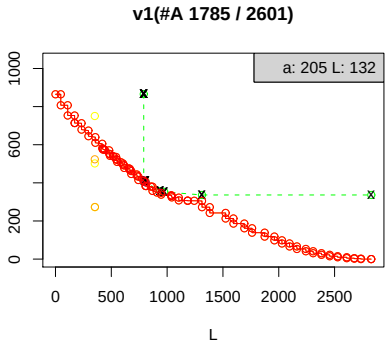
```
# Solve the models for perfect and imperfect monitoring
m.pm = solve.game(m.pm)
```

```
m = solve.game(m, ignore.ae = m$action.val[,1]>m$action.val[,2])
```

The last line solves the model with imperfect monitoring. The parameter `ignore.ae` of the function `solve.model` can be used to ignore certain action profiles as candidates for optimal equilibrium action profiles.[10] Since the stage game is symmetric, we can assume w.l.o.g. that player 1 produces the same amount or weakly more than player 2 on the equilibrium path. Ignoring the action profiles where player 2 produces more, speeds up calculation. Still, it takes some seconds to solve the model.

The graphical illustration of the solution process

During the process the toolbox shows you some dynamic graphical representation of the solution process. A screen-shot is shown below:



The title first shows the actual state that is solved: v1 stands for player 1's punishment state. In brackets we find the total number of action profiles that have to be checked (the second number are all action profiles, the first number is the number of action profiles who have a best-reply payoff below some Nash equilibrium of the stage game). The grey box on the top-right corner shows how many action profiles have already been investigated (205), and for how many of those action profiles one or more linear programs had to be solved (here 132). The remaining 205-132=72 action profiles had been quickly dismissed as candidates for optimal punishment profiles by using the methods described in the appendix of our theoretical paper. The red points and line show the lower envelope of player 1's punishment payoffs $v_1(L)$ from the 205 action profiles that have already been solved. The green line shows the relevant part of the function $v_1(L|a)$ from

[10] For games with imperfect public monitoring, the parameters ignore.ae or ignore.ai can play similar roles than the link functions link.fun.ae and link.fun.ai used in the analyze of repeated games with perfect monitoring with large action spaces (see Sections 5 and 6). The difference in the two approaches is due to the fact that for games with imperfect monitoring, we always have to specify the whole grid of action profiles while for games with perfect monitoring we can use an algorithm that locally refines grids.

the last action profile a that possible could improve the envelope. The yellow and orange dots indicate action profiles a' that were inspected after a and could be dismissed before $v_1(L|a')$ had to be calculated.[11] Towards the end of the computation, you will see large clouds of orange dots, which indicate that many action profiles can be dismissed quickly. Admittedly, these plots are not very artful, but I felt that some visualization is helpful to get a feeling of what is going on.

If during the solution you click on the text window, you can see reports from the linear programming solver. You may also get some warnings that some of the many conducted linear optimization problems was plaqued by numerically instability of no solution could be found. The program then tries a different method (like dual simplex) and the message should tell you whether that worked. If the warning persist, you have to decide whether you can ignore the warning or not (the description of the actual linear optimization problem where the warning occured can be found above the warning). For example, it can well be the case that you have a stage game where some action profiles can never be implemented.

To compare the equilibrium payoffs under imperfect and perfect monitoring, we can type

```
plot.compare.models(m.pm,m,xvar="delta",yvar="Ue",
        m1.name = "Cournot",m2.name="Green-Porter Style",
        legend.pos ="bottomright")
```

The resulting plot is shown in Figure 13.

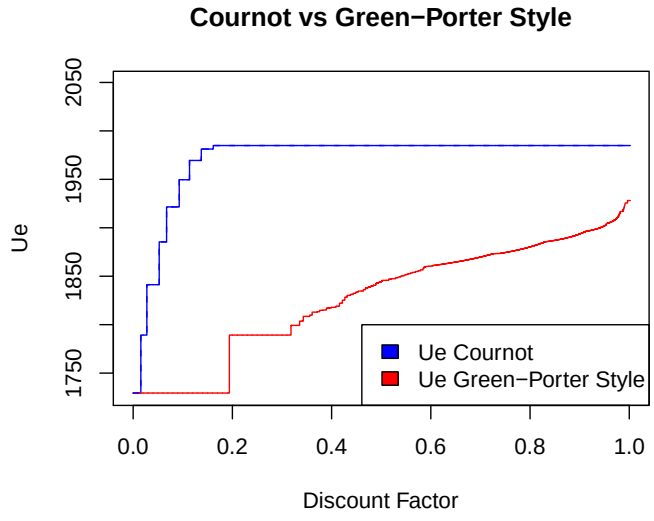


Figure 13: Joint equilibrium and punishment payoffs of the modified Green-Porter model and the case with perfect monitoring

We see how much cooperation can be facilitated if firms can perfectly monitor each others actions. It is thus not-surprising that in reality effective monitoring technologies are crucial for the stability and profitability of cartels.

For the case of imperfect monitoring, the joint monopoly profit cannot be achieved by the cartel even as the discount factor δ approaches 1, This is a well known result. Formally, the folk theorem does not hold because the full dimensionality criterion of Fudenberg, Levine and Maskin (1994) is not satisfied if firms only observe price signals. A rough intuition is that market prices contain no information about which firm potentially deviated from the agreement. Realizations of low market prices therefore have to be punished with jointly inefficient continuation play.

If you type `m$opt.mat`, you see the structure of optimal action plans for all discount factors. The list is quite long and therefore not easy to comprehend. To get some graphical representation type:

```
ret = levelplot.rep(m=m,z=m$G,main="G and opt. action plan",
  focus=2,cuts=100, xlab="q1",ylab="q2", identify=NULL,
  col.nash="purple",col.br1 = "green", col.br2 = "greenyellow",
  col.ae = "blue", col.a1 = "red")
```

Which draws a levelplot of maximal stage game payoffs and adds some information about the optimal action plans. The resulting plot is shown in Figure 14.

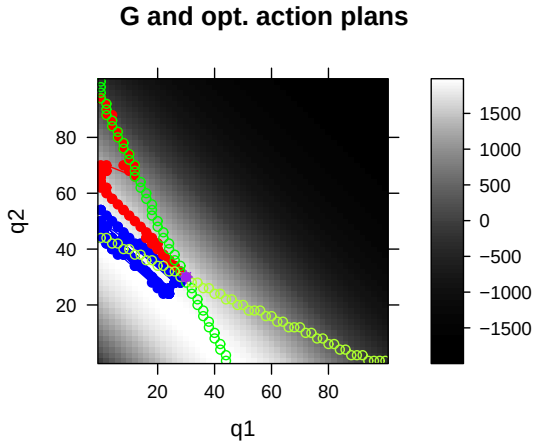


Figure 14: Levelplot of joint stage game payoffs and information about optimal action plans

The green line show player 1 and 2's best reply functions in the stage game and the purple dot the Nash equilibrium of the stage game. The blue dots show the optimal equilibrium action profiles for different discount factor and the red dots depict optimal punishment profiles. The order of the profiles can be somewhat disentangled by the lines that connect the different optimal profiles. The nearer a profile is to the stage game Nash equilibrium, the lower is the discount factor for which the action profile is optimal.

8.4 Optimal equilibrium state action profiles

Let us investigate in more detail the optimal equilibrium state action profiles. To get a zoomed picture of the relevant range of actions, paste the following code:

```
ret = levelplot.rep(m=m,z=m$G,main="G and opt. ae",
  focus=2,cuts=100,xlab="q1",ylab="q2", identify="Ue",
  xlim=c(-1,35),ylim=c(20,56), zlim=c(500,2000),
  col.nash="purple",col.br1 = NA, col.br2 = NA,
  col.ae = "blue", col.a1 = NA)
```

The parameters `xlim` and `ylim` specify the range of player 1 and player 2's action levels for which the plot shall be drawn. The parameter `zlim` can be used to reduce the range of over which the grey scale for G will be defined. By setting `col.br1 = NA`, the stage game best replies of player 1 will not be drawn (similar for the other parameters). The parameter `identify="Ue"`, allows you to interactively click on the levelplot to select specific action profiles and compare the form of the function $U^e(L|a)$ for the different selected points. Clicking once on a point opens a new graphic window that shows this plot (the new graphic window may open below the actual window, you may have to move the current window to see the new one). Note that the R prompt does not take any commands while in the point selection mode. To leave this mode, press the right mouse button on the levelplot and say stop or press the red stop button in your R Gui. Figure 15 illustrates the level plot and the function $U^e(L|a)$ for 4 selected points:

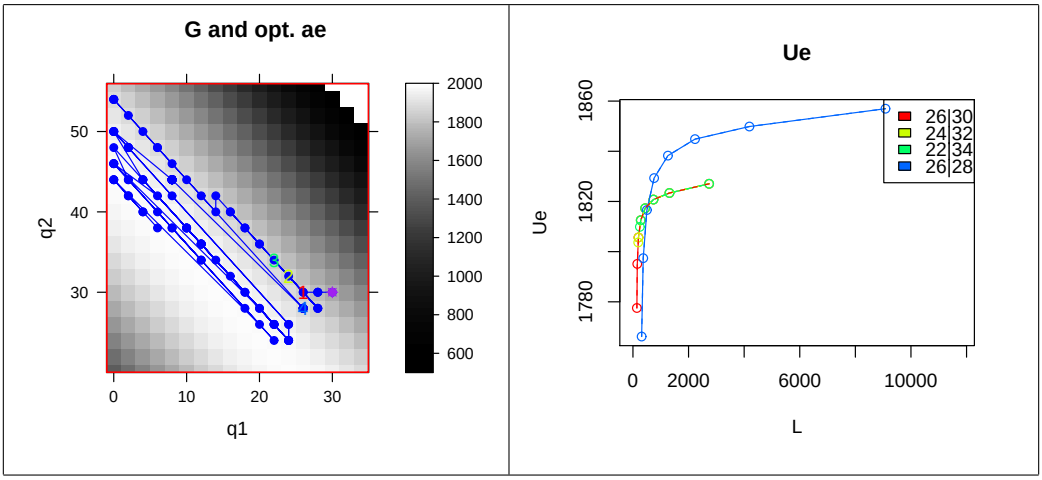


Figure 15: Investigating the structure of optimal equilibrium action profiles a^e

While the fourth point selected point has a joint output of $26 + 28 = 54$, the first 3 selected points lie on a line that have the same joint output $q_1 + q_2 = 56$ and thus also the same joint stage game payoff $G(a)$. The right hand side of Figure 15 confirms that they also have the same functional form for $U^e(L|a)$ with the exception that the more symmetric action profiles have a lower liquidity requirement. (It is easier to see this if you click yourself!).

You probably asked yourself the following question: What is the reason that the equilibrium points wiggle around like they do and why are so often asymmetric action profiles optimal? I below I suggest a list of possible answers (the plot in the right hand side of Figure 15 may already suggest which ones are more likely to be true):

1. There is an error either in the way the model is set-up or in the toolbox. (While the toolbox already went through quite some debugging, it is still in development stage and errors are well possible.)

2. For a given discount factor, several action profiles can be optimal. The algorithm does not necessarily pick the most symmetric ones for every discount factor and the wiggling around may simply be due to indifference.

3. The reason for the wiggling around is connected to the fact that we consider a finite approximation of the originally continuous stage game. It could be the case that in the continuous game, fully symmetric action profiles are optimal but not in the finite approximation. For example there can exist some asymmetric profiles that correspond to certain level of average outputs $\frac{q_1+q_2}{2}$, but on the finite grid no symmetric profile with the same average output exists.

4. Even in the continuous version of the game, it is an inherent characteristic that asymmetric action profiles are optimal on the equilibrium path. For example, one might think that perhaps the asymmetric incentives to deviate under an asymmetric equilibrium profile might allow for more efficient asymmetric punishments if low prices are observed.

In the end one has to resolve the question whether asymmetric equilibrium action profiles can be optimal by a theoretical proof, but the toolbox can be helpful in making sensible conjectures and get intuition which result might be more likely to be true and why that would be the case. The plots in Figure 15, are so far still suggesting that in a continuous version of the game, symmetric profiles might always be optimal (in particular, since they have a lower liquidity requirement).

To get more insight you can click around a bit more or paste the following code:

```
ai.mat = get.ai.mat(m)
ignore.ae = !(ai.mat[,1]==ai.mat[,2] |
               ai.mat[,1] == (ai.mat[,2]-1) )
m.sym.ae = solve.game(m,ignore.ae=ignore.ae)
```

This code solves the model with the restriction that only symmetric action profiles (and profiles that are one action unit left of the main diagonal)[12] are

[12]We add these shifted profiles to account for the effect described under point 3: off-diagonal elements can have total contribution levels that in the discretized game are not obtained by any fully symmetric profile.

considered for equilibrium action profiles. The matrix `ai.mat` simply returns the index of each players actions (in comparison to `m$action.val.mat`, which returns the action value, i.e. the quantity). You can investigate payoffs and optimal action plans of the restricted solved model using the methods described above. The following command allows direct comparison of the maximal equilibrium payoffs of the restricted and unrestricted models:

```
plot.compare.models(m,m.sym.ae,xvar="L",yvar="Ue",
   m1.name = "unrestricted", m2.name = "sym. ae",
   legend.pos  =  "bottomright",identify="Ue")
```

The resulting plot is shown in Figure 16. One can almost see no difference in the maximal joint payoffs for the restricted and unrestricted case.

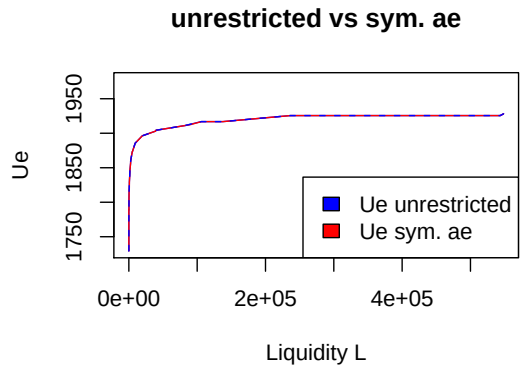


Figure 16: Comparison of equilibrium payoffs solution with symmetric equilibrium actions and unrestricted solution

Small differences can be due to inaccuracies in calculations with finite precision floating point numbers.[13]

The example thus supports the following conjecture:

Conjecture. *In the symmetric Green-Porter style model with side-payments and continuous action and signal spaces, every pure strategy PPE payoff can be implemented with stationary equilibria that use symmetric equilibrium action profiles.*

I admit that this conjecture is not very surprising, in particular given the results from Abreu, Pearce, Stacchetti (1986)... but I wanted to illustrate how the toolbox can help you to form quite educated guesses.

[13]My implementation of the algorithm to calculate upper envelopes only guarantees to calculate the upper envelope within some tolerance interval. I introduced the tolerance level to make the calculations more robustness with respect to inaccuracies of calculations with floating point numbers.

8.5 Punishment States

Let us now have a closer look at the structure of optimal punishment profiles. In Abreu, Pearce & Stacchetti (1986) optimal punishment profiles will be symmetric, but this won't be the case in our set-up where we allow for observable monetary transfers and money burning. This means we can use asymmetric punishments in case a player defects from a required payment. The following code generates the zoomed levelplot shown in Figure 17 (left):

```
# Level plots for the punishment state
ret = levelplot.rep(m=m,z=m$c[,1],main="c1 and opt.a1",
  focus=0,cuts=100,
  xlab="q1",ylab="q2", identify="v1",
  xlim=c(-1,35),ylim=c(25,100),
  col.nash="purple",col.br1 = "green", col.br2 = "greenyellow",
  col.ae = NA, col.a1 = "red")
```

Let us interpret the resulting levelplot. Unless discount factors are so low that the optimal punishment profile a^1 is the Nash equilibrium of the stage game, firm 2 chooses an output level above the Nash equilibrium output and above its best-reply output. This makes perfectly sense, since firm 1 is more severely punished if firm 2 chooses a high output, or more precisely, firm 1's best-reply payoff is the lower the higher is firm 2's output. Firm 1 always chooses an output weakly below his Nash equilibrium output. For intermediate discount factors firm 1's output is below its stage game best-reply output. However, there is a critical discount factor so that for higher discount factors, firm 1 plays a best-reply in its punishment profile.

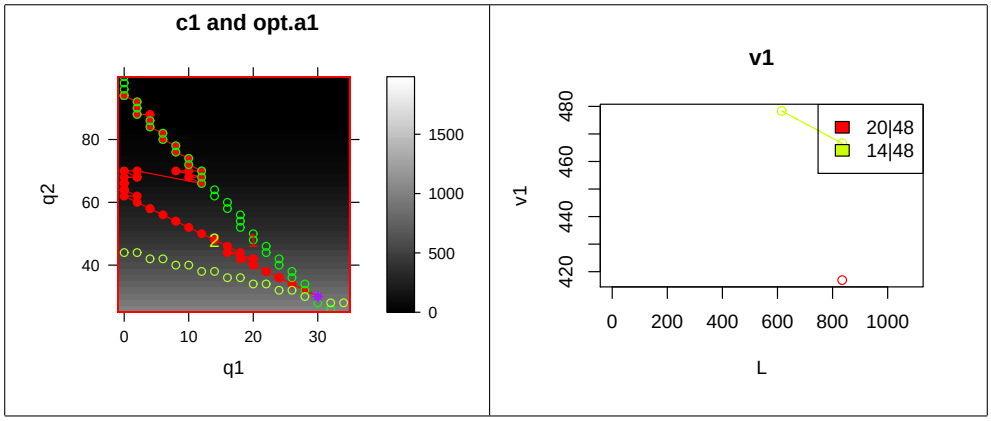


Figure 17: Analyzing the optimal punishment profiles for player 1 in the Green-Porter Style Model

One may wonder whether this structural break is indeed an inherent characteristic of optimal punishment profiles or whether similar to the equilibrium action profiles there is a broad range of optimal punishment profiles for every discount factor and the algorithm just by chance turned out to have selected a path with this peculiar structural break. By clicking with the left mouse

button on the level plot I selected two action profiles marked with 1 and 2 in the level plot. In both action profiles player 2 chooses an output of 48: in the first action profile player 1 plays a stage game best-reply to 48 and the second action profile is an optimal punishment profile for player 1. The right hand side of Figure 17 shows the opened plot in which the function $v_1(L|a)$ are drawn for the two selected points. We find that the second profile (the optimal action profile) has a lower liquidity requirement but the first profile (player 1 plays a stage game best-reply) can implement lower punishment payoffs.

The latter observation illustrates Lemma 2 in our theoretical paper: player 1's cheating payoff $c_1(a^1)$ is the lowest punishment payoff that can ever be implemented with a given punishment profile a^1. This punishment payoff can always be attained if player 1 plays a stage-game best reply. Generically, punishment payoffs are strictly above $c_1(a^1)$ if player 1 does not play a stage game best-reply in his punishment profile.

That the first action profile has a higher liquidity requirement than the second action profile may be explained by the following intuition: Since quantities are strategic substitutes, a high output of firm 2 can be easier sustained if the punished firm 1 chooses a low output. There seems to be some convexity in the total liquidity requirement in the sense that larger absolute deviations from a firms' best reply are increasingly expensive and that therefore points somewhat centered between the two best-reply functions can achieve an optimal trade-off between liquidity requirement and reducing player 1's punishment payoff. The following code generates the plot in Figure 18 that shows graphically how liquidity requirements are lower between the two best reply-functions (levels shown in blue correspond to action plans for which the liquidity requirement has not been calculated when the model has been solved).

```
# Show liquidity requirements
ret = levelplot.rep(m=m,z=m$L,main="L",focus=-3,cuts=100,
  xlab="q1",ylab="q2", identify="v1",
  xlim=c(-1,35),ylim=c(25,100),
  col.nash="purple",col.br1 = "green", col.br2 = "greenyellow",
  col.ae = NA, col.a1 = NA)
```

When the structural break occurs, firm 1 already produces an output of 1 (lower outputs are forbidden by the non-negativity constraints). For larger discount factors lower punishment payoffs seem to be more easily implementable in a regime where player 1 plays a best-reply. I think this makes intuitive sense. The following code can be used to generate Figure 19, which verifies that restricting player 1 to play a best-reply in his punishment would indeed yield lower punishment payoffs:

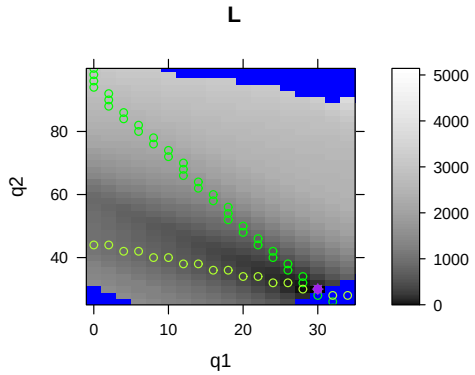


Figure 18: Liquidity requirements

```
ignore.a1 = (m$g[,1] != m$c[,1])
m.a1.br1 = solve.game(m,ignore.a1=ignore.a1)
plot.compare.models(m,m.a1.br1,xvar="L",yvar="v1", x.max=4000,
                    m1.name="m", m2.name="m1.a1.br1")
```

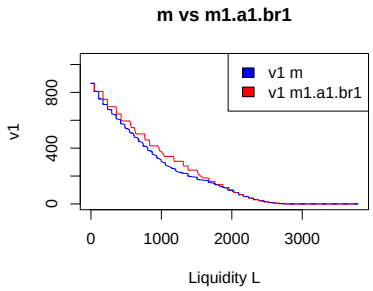


Figure 19: Punishment payoffs in optimal model and under restriction that firm 1 plays a stage game best reply.

For intermediate levels of liquidity, strictly lower punishment payoffs can be implemented if the set of player 1's punishment profiles is not restricted to action profiles where player 1 plays a best reply.

8.6 Exercises

Exercise 8.2. (Optimal punishment profiles if production costs are very low) Solve a variant of the studied oligopoly in Section 6.2 where firms have no production costs. Draw a levelplot to illustrate the structure of optimal punishment profiles. Explain the result intuitively. Now study the structure of optimal equilibria for a sequence of games where -ceteris paribus- you slightly increase the constant marginal production costs. Perform the same analysis for the case of perfect monitoring.

Exercise 8.3. (Symmetric punishment profiles) Solve the model under the restriction that only symmetric punishment profiles with $q_1^i = q_2^i$ can be used (use the ignore.ai parameter). Why would you think that the resulting payoff set is the same than under optimal contracts without monetary transfers studied by Abreu, Pearce and Stachetti (1986)?

Also solve the case of perfect monitoring with a restriction to symmetric punishment profiles (you can use the parameter ignore.ai in a similar fashion as for games with imperfect monitoring or use the framework with link functions described in Section 5.3) Why is the resulting equilibrium set larger still larger than in Abreu's (1986) analysis of optimal strongly symmetric equilibria in repeated Cournot games with perfect monitoring?

Exercise 8.4. Harrington & Skrzypacz (2007) study a collusion model where firms can make observable monetary transfers. The structure of the model is inspired by actual behavior of revealed cartels, most notably the Lysine and Citric Acids Cartels.[14] Harrington & Skrzypacz do not consider optimal punishments, but assume that if a firm fails to make required monetary transfers the punishment is that all firms revert to infinite repetition of the stage game Nash equilibrium.

Solve our model with the restriction that only the stage game Nash equilibrium is allowed as punishment profile. Note that the resulting punishment paths are not equal to an infinite repetition of the stage game Nash equilibrium: they have a stick and carrot structure where the stage game Nash equilibrium is played only for one period and one reverts back to the equilibrium state if all required payments are made after this punishment period. Why are the resulting punishment payoffs nevertheless the same as if one would use a grim-trigger punishment á la Harrington & Skrzypacz?

Actually, Harrington & Skrzypacz did not consider a framework with quantity setting, but price setting firms and stochastic shocks to realized market shares. Implement (a discrete approximation) of their game using the toolbox and analyze the structure of optimal equilibria and resulting payoff sets.

Exercise 8.5. (Sensitivity analysis of discrete approximations) Analyze how sensitive the resulting payoff sets are to the chosen discrete approximations of the action and signal space. Perform comparative statics with respect to the number of actions and signals that are used to approximate the continuous game.

9 Auditing

This Section just contains some first results on an idea I am currently working on. It shall illustrate how our theoretical results and the toolbox can be used to compare in more detail different monitoring technologies that players may use to structure their relationship.

[14] Even if you are not an expert on collusion, you might have heard some details from the Lysine Cartel from the movie The Informant. The movie is based on Kurt Eichenwald's great book with the same name that documents the FBI investigations on that case.

In a repeated oligopoly with perfect monitoring firms can publicly observe each others sales. Such public signals could, for example, be created by publicly verifiable reports on a firm's sales from an external auditor. Indeed, we now from the FBI tapes of the Lysine Cartel that the conspiring firms, where indeed considering to use external auditors to verify sales reports across the cartel members. Auditing may not only be relevant for collusion, but also in joint team production where under normal circumstances only the resulting success or failure of the project is observable, one may use audits to get a public signal about each players effort choices. Such audit reports may not be evidence enough to base court-enforceable contracts upon them, but they may be relevant for the selected continuation equilibrium. As auditing is expensive, one may not want to audit every period, but rather rely on infrequent, randomly triggered auditing.

Here I want to present a first simple framework that allows to augment any repeated game with imperfect public monitoring by a simple auditing structure. There shall be an external auditor that performs after each period with a fixed probability π an audit on all players. An audit shall perfectly reveal the chosen actions of all players an make it commonly known among the players.

There is no conceptual difficulty to extend any game with imperfect monitoring by such an auditing scheme. We simply have to extend the old signal space Y to a new signal space $\tilde{Y} = Y \times A \cup \{\emptyset\}$. A new signal is given by $\tilde{y} = (y, \alpha)$ where y indicates the normal signal and $\alpha \in A$ is the action profile reported by the audit and $\alpha = \emptyset$ indicates that no audit took place. The distribution of the new signals is then simply given by

$$\tilde{\phi}(y, \alpha|a) = \begin{cases} \phi(y|a) * (1-\pi) & \text{if } \alpha{=}\emptyset \\ \phi(y|a) * \pi & \text{if } \alpha{=}a \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

The only practical drawback is that the signal space can now become very large if the original game already had a large signal space. Computation time may substantially increase.

However, there are simply methods to reduce computational complexity. Let me explain some of them. Since, we have assumed that auditing is perfect and always fully reveals the played action profile, the audit report is a sufficient statistic for actual behavior, i.e. one can ignore the signal y if an audit took place. This means that the modified game with signal space $\hat{Y} = Y \cup A$ and signal distribution

$$\hat{\phi}(\hat{y}|a) = \begin{cases} \phi(\hat{y}|a) * (1-\pi) & \text{if } \hat{y} \in Y \\ \pi & \text{if } \hat{y} = a \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

has the same payoff sets than the game specified by $\tilde{Y}$ and $\tilde{\phi}$but considerably less signals.

We can reduce computational complexity even further. Since an audit is perfect, it only matters whether some deviation took place or not. It is always optimal that a player that was caught to have deviated is required to pay the maximum amount he can be induced to be paid $\frac{\delta}{1-\delta}(u_i - v_i)$, independent to which action he did deviate. In the static problems that we use to characterize the repeated game, we can correspondingly require player i to make the maximum payment $\lambda_i L$ if he was found to have unilaterally deviated from the required action profile. This can be incorporated by a simple modification of the action constraints in the static problem. The new action constraints with auditing become:

$$g_i(a) - (1-\pi)E_y[p_i|a] \geq g_i(\hat{a}_i, a_{-i}) - (1-\pi)E_y[p_i|\hat{a}_i, a_{-i}] - \pi\lambda_i L \quad (6)$$

My toolbox allows to directly implement stochastic auditing by modifying audit probabilities in the way shown above. To analyze a model m with audit probability of say 20%, simply enter the following line

```
m = set.audit.prob(m,0.2)
```

after you have initialized the model, but before you solve it. Here is an example that compares the effects of auditing in Green-Porter Style models:

```
# Initialize models
m.pm = init.payoffs.green.porter()
m.imp = init.signals.green.porter(m.pm)
m.10 = set.audit.prob(m.imp,0.1)
m.imp$name = "Green-Porter (no audits)"
m.pm$name = "Green-Porter (permanent audits)"
m.10$name = "Green-Porter (10% audits)"
# Solve models using symmetric action profiles on the
# equilibrium path
ai.mat = get.ai.mat(m)
ignore.ae = !(ai.mat[,1]==ai.mat[,2] |
ai.mat[,1] == (ai.mat[,2]-1))
m.10 = solve.game(m.10 ,ignore.ae=ignore.ae)
m.pm = solve.game(m.pm ,ignore.ae=ignore.ae)
m.imp = solve.game(m.imp,ignore.ae=ignore.ae)

# Compare the models graphically
plot.compare.models(m.10,m.imp,xvar="delta",yvar="Ue",
m1.name = "10% audit", m2.name = "no audits",
legend.pos ="bottomright")

plot.compare.models(m.10,m.pm,xvar="delta",yvar="Ue",
m1.name = "10% audit", m2.name = "100% audits",
legend.pos ="bottomright")
```

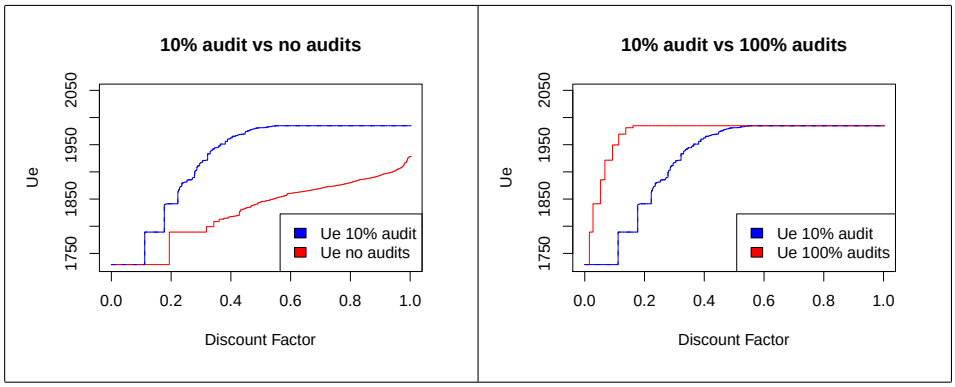


Figure 20: Collusion with different auditing probabilities

The resulting graphs are shown in Figure 20:

We find that an audit in 10% of periods can already substantially increase the scope for collusion. Thus having a perfect, incorruptible auditor can be very valuable. The following exercise invites you to study more of the effects of auditing:

Exercise 9.1. Pick one game with imperfect public monitoring, e.g. the Noisy Prisoners' Dilemma game or some Green-Porter model. Also pick a finite grid of auditing probabilities and perform graphical comparative statics of maximum equilibrium payoffs with respect to the auditing probability π (see Section 4.2 for how to perform such comparative statics). Assume that auditing all firms in one period involves costs of $M > 0$, so that expected equilibrium payoffs that account for auditing costs are $U - \pi M$. Find the optimal auditing probability (from your grid) for every discount factor.

What I explained above, is just a very simple framework. I am working on a more extended set-up that allows to condition audit probabilities on the realized signals and explicitly gives players the opportunity to refuse to be audited or refuse to pay any auditing costs.

Part III
Appendices

10 Some Hints for Debugging

When I want to study a new class of games, it only seldom happens that the code runs on the first try without any errors. At least when I constructed some new example there always was some mistake in the beginning. Furthermore, the toolbox itself is in a very early stage and may still contain errors. Also the toolbox is not very user friendly when you initialize a game with some incompatible parameter constellation. Most likely, you will get some quite incomprehensible error message.

In this Appendix, I want to illustrate, how to find and correct an error. Consider the following flawed version of the function from Section 4.3 that shall initialize and solve an n-player public goods game:

```
pg.game = function(n,X,k=rep(((1+1/n)/2),n)) {
  store.objects("pg.game")
  #restore.objects("pg.game")

  g.fun = function(x.mat) {
    store.objects("g.fun")
    #restore.objects("g.fun")
    g = matrix(0,NROW(x.mat),n)
    xsum = rowSums(x.mat)
    for (i in 1:n) {
      g[,i] = xsum / n - k[i]*x.mat[,i]
    }
    g
  }
  name=paste(n,"Player Public Goods Game")
  m = init.game(n=2,g.fun=g.fun,action.val = X,
                      name=name, lab.ai=round(X,2))
  m=solve.game(m)
  plot(m)
  m
}
```

When I call the function, I get an error:

```
m = pg.game(n = 3, X = 0:10, k = c(0.4, 0.6, 0.8))

## Error: subscript out of bounds
```

The error message is probably not very informative for you. To get more insights, were the error occurred, we can make the following function call:

```
traceback()

   ## 3:  g.fun(val.mat)
   ## 2:  init.game(n = 2, g.fun = g.fun, action.val = X, name = name,

   ## lab.ai = round(X, 2))
   ## 1:  pg.game(n = 3, X = 0:10, k = c(0.4, 0.6, 0.8))
```

At the top of the resulting list, you see the function where the error has occurred, below you find the remaining call stack, i.e. the functions from which that function has been called. (Probably, you already saw the error, but please pretend not to, so that I can explain how a blind me would proceed debugging.)

From the traceback, we know that the error occurred somewhere in the function `g.fun`, but where exactly? R offers several methods for debugging that allow to execute the commands in a function step by step. These debugging facilities are explained, e.g. in `http://cran.r-project.org/doc/manuals/R-exts.pdf` (Section 4) or in `http://cran.r-project.org/doc/Rnews/Rnews_2003-3.pdf`.

Personally, I prefer a slightly different approach that I already briefly discussed in the hints on R programming in Section 4. At the beginning of almost all functions that I create, I add the lines

```
store.objects("function.name")
#restore.objects('function.name')
```

The first line stores all the local variables of the function in a global variable container.[15] The second line is commented, i.e. it will not be run. However, if you want to debug the function, you can paste that line and all stored parameters from the last time the function was called, will be copied in the global environment. You can then simple paste the remaining code of the function and see where the error occurred. Let us apply this procedure with `g.fun`:

```
restore.objects("g.fun")
restore.objects("pg.game")
g = matrix(0, NROW(x.mat), n)
xsum = rowSums(x.mat)
for (i in 1:n) {
    g[, i] = xsum/n - k[i] * x.mat[, i]
}
```

```
## Error: subscript out of bounds
```

Ah... we get an error in the for loop. Practically, the integer `i` will keep the value it had when the error was thrown... This means we can simple paste the interior of the for loop (or parts of the commands) to investigate further where the error has occured. Let us thus proceed the investigation by typing the following sequence of commands:

```
i
```

```
## [1] 3
```

```
g[, i] = xsum/n - k[i] * x.mat[, i]
```

```
## Error: subscript out of bounds
```

```
n
```

```
## [1] 3
```

[15] All variables means all local variables that are known when `store.objects` is called. If, as I usually do, `store.objects` is called at the beginning of a function only the arguments of the function call are stored.

What does that mean? `n=100` ??? That does not make any sense at all... OK, the reason for this apparently strange result is connected to the Advanced Programming Hints, I gave in Section 4.3 and illustrates one weakness in my approach to debugging. The function `g.fun` uses the variable `n` from its parent function `pg.game`. As `n` was not an argument of `g.fun`, the function call `store.objects("g.fun")` did not store the variable `n`, i.e. the variable `n` was not restored properly by `restore.objects("g.fun")` and it just kept the value it had from some previous assignment. To solve the problem, we also have to call

```
restore.objects("pg.game")
```

in order to restore the local variables of the last call to `pg.game` (this assumes that `pg.game` has not changed the parameter n in its function body, which is the case).

Having done this, let us paste again the inner code of the function and then continue the search for the reason of the error:

```
n
```

```
## [1] 3
```

```
g[, i]
```

```
##   [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [36] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [71] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [106] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
k[i]
```

```
## [1] 0.8
```

```
x.mat[, i]
```

```
## Error: subscript out of bounds
```

```
dim(x.mat)
```

```
## [1] 121   2
```

```
i
```

```
## [1] 3
```

Aah, the culprit is the matrix of action values argument `x.mat`! The matrix of action values has only two columns, even though I have told the function

`pg.game` that it shall set `n=3`, i.e. it shall initialize a three player game. So let us investigate the function that calls `g.fun`. Recall the traceback:

```
3: g.fun(val.mat)
2: init.game(n = 2, g.fun = g.fun, action.val = X, name = name,
       lab.ai = round(X, 2))
1: pg.game(n = 3, X = 0:10, k = c(0.4, 0.6, 0.8))
```

This means it was `init.game` that called `g.fun`. But `init.game` is a function of the toolbox, not a user defined function. This is no problem, as there are different ways how you can access the source code of a function of my toolbox. One way is to simply type the function name (without any brackets or arguments) in the R prompt:

```
init.game
# Output ommitted in the PDF file
```

The whole code of function will then be displayed in your console. You can simply copy and paste it into your text editor and from there past the desired pieces into your code. The drawback is that in the version you see, all comments have been removed (inclusive the commented line `#restore.objects("init.game")`)An alternative way is to find the function directly from the source files of the toolbox that you can find on my homepage. These source files still contain all comments.[16]

Most of my own functions start with a call to `store.objects`, i.e. you can debug them in the way described above. Let us do so, i.e. we paste the code from `restore.objects` up to the call to `g.fun`:

```
## Error: could not find function "repgames.startup"
## Error: subscript out of bounds


restore.objects("init.game")

m = list()
class(m)=c("repgame","list")

#[... a lot of more code, omitted here...]


# Initialize ai.dim
if (gtype=="g.mat")  {
  m$g = g
```

[16] My source code is distributed across different .r files. If you use the Crimson Editor, the feature "Find in files..." in the menu "Search" is helpful to find in which file a certain function is defined.

```
} else if (gtype=="g.fun") {
  val.mat = make.grid.matrix(x=action.val.list)
  m$g = g.fun(val.mat)
} else if (gtype=="g1") {
  m$g1 = g1
  m$g2 = g2
  # Table that contains all payoffs nA rows and n columns
  m$g = cbind(as.vector(t(g1)),as.vector(t(g2)))
}
```

```
## Error: subscript out of bounds
```

The error took place in this if structure.... Let us dissect it further:

```
gtype
```

```
## [1] "g.fun"
```

```
val.mat = make.grid.matrix(x=action.val.list)
m$g = g.fun(val.mat)
```

```
## Error: subscript out of bounds
```

```
val.mat[1:3,]
```

```
##      [,1] [,2]
## [1,]    0    0
## [2,]    0    1
## [3,]    0    2
```

```
action.val.list
```

```
## [[1]]
##  [1]  0  1  2  3  4  5  6  7  8  9 10
##
## [[2]]
##  [1]  0  1  2  3  4  5  6  7  8  9 10
##
```

OK... the matrix `val.mat` has only 2 columns, and the list `action.val.list` only 2 vector elements. But I did initialize a three player game, or didn't I. Let's check...

```
n
```

```
## [1] 2
```

In the function `init.game` the number of players `n` is set to 2. Why is that... Let us look where we call `init.game` in `pg.game`.

```
m = init.game(n=2,g.fun=g.fun,action.val = X,
                      name=name, lab.ai=round(X,2))

## Error: subscript out of bounds
```

Finally, the mistake is detected! We wrongly set `n=2` when calling `init.game`. Such mistakes can easily happen if one builds code by copy-and-pasting it from different examples. Correcting the bug is quickly done, by changing the line in `pg.game` to:

```
m = init.game(n=n,g.fun=g.fun,action.val = X,
                      name=name, lab.ai=round(X,2))
```

Now everything should work fine.

It looks like of a lot of effort to find a small typo, but that is just the way it often is. Personally, I think that with a bit of experience debugging can be quite fun: it is a bit like an interactive detective story between you and the computer. Furthermore, bugs are a good commitment device that force you to understand in detail how some piece of code works. The word "Interactively" in the title of this paper can well be referred to the interactive process of debugging the code that specifies your repeated game. It is a bit as if you had somebody that reads a proof of your paper and gives you a cryptic error message if something is wrong and step by step you have to encrypt what went wrong (who knows, maybe programming the computer implementation of your game may indeed help you to find a mistake in some proof).

Unfortunately, not every mistake reveals itself through an error message. So you also have to check whether the results of the solved model look sensible. Ideally, you know the solution for some special case and can first test the program on that case. I would definitely recommend to extensively check whether the functions that describe stage game payoffs and cheating payoffs work correctly. Remember the tools to check cheating payoffs described in Section 6.2. Also note that plotting is often a very powerful tool for debugging.

Modifying the source code of the toolbox

As the toolbox is very young, it is very likely that it still contains several bugs. Above I briefly discussed, how you can trace through the source code of my package. If you find an error in my toolbox, please send me an email (skranz@uni-bonn.de). I will try to correct the error, but I cannot guarantee that I will be able to do it quickly.

So maybe you prefer to correct the error yourself. To do this, you have to download the source code from my homepage and change it appropriately. Building an R package from the changed source code is quite an

endeavor, however. I rather suggest the following method (described for windows): Create a link of the R program that uses as working folder the folder that contains the source code. Then open R via this link. Instead of calling `library(repgames)`, paste the content of the file `rep_loadfiles.r` into your console. This will manually load the actual version of the sources. Any modifications that you made to the source files will be included.

Debugging warnings

Sometimes the toolbox does not throw an error that terminates the execution, but only a warning that sounds uncomfortable. Paste

```
options(warn=2)
```

to tell R that it shall stop after any warning, i.e. warnings are treated like errors. You can then debug in the way explained above. To reset to the normal mode, call

```
options(warn=0)
```

If you want R to stop only after some specific warning, you have to locate the function that throws the warning in my source code and add after the line that throws the warning a line `stop()` and reload the modified source code in the way explained in the previous box.

11 References

Abreu, Dilip. 1986. "Extremal equilibria of oligopolistic supergames", Journal of Economic Theory, vol 39 (1): 191-225.

Abreu, Dilip & Pearce, David & Stacchetti, Ennio, 1986. "Optimal cartel equilibria with imperfect monitoring," Journal of Economic Theory, vol. 39(1): 251-269.

Abreu, Dilip, 1988. "On the Theory of Infinitely Repeated Games with Discounting," Econometrica, 56, 383-396.

Abreu, Dilip & Pearce, David & Stacchetti, Ennio, 1990. "Toward a Theory of Discounted Repeated Games with Imperfect Monitoring," Econometrica, 58, 1041-63.

Fudenberg, Drew & Levine, David & Maskin, Eric, 1994. "The Folk Theorem with Imperfect Public Information," Econometrica, 62, 997-1039.

Kranz, Sebastian & Ohlendorf, Susanne, 2009. "Renegotiation-Proof Relational Contracts with Side Payments"

Goldluecke, Susanne & Kranz, Sebastian, 2012. "Infinitely Repeated Games with Imperfect Public Monitoring and Monetary Transfers", Journal of Economic Theory (forthcoming).

Green, Edward J & Porter, Robert H, 1984. "Noncooperative Collusion under Imperfect Price Information," Econometrica, Econometric Society, vol. 52(1): 87-100.

Harrington, Joseph E. & Skrzypacz, Andrzej Jr., 2007. "Collusion under Monitoring of Sales", The RAND Journal of Economics, vol. 38(2):314-331.

Isaac, R.M.,Walker, J.M., 1988., "Group Size Effects in Public Goods Provision: The Voluntary Contributions Mechanism," Quarterly Journal of Economics 103(1): 179-199.

Kranz, Sebastian, 2010. "Moral Norms in a Partly Compliant Society," Games and Economic Behavior, vol 68 (1): 255-274.