HyRiM: A Package for Multicriteria Game Theory over the Space of Probability Distributions

Stefan Rass
Universität Klagenfurt

Sandra König
Austrian Institute of Technology

Abstract

Zero-sum normal form games usually come as matrices defining bilinear real-valued functions, for which saddle-point values are sought. This problem is convertible into a linear programming problem, and as such solvable up to large scale. Such games have become popular tools in security and risk management, where they provide powerful worst-case models of defense against rationally acting adversaries. A practical difficulty in security is herein commonly the definition of payoffs, which for risk management, are mostly based on empirical data. Moreover, optimizing security requires the defender to find a balanced defense regarding several security goals against one (or many) adversaries. Thus, *security risk management games* are challenging in several ways, since (i) their payoffs may be defined only from empirical data, (ii) the data upon which the model rests is uncertain, and (iii) the optimization is over multiple criteria that may have interdependencies (up to conflicts). The **HyRiM** package, available from CRAN, supports parts of the game-theoretic risk management approach by lifting the theory of two-player zero-sum games from real-valued payoffs to payoff functions whose values are probability densities, and playing multi-objective zero-sum games over stochastic orders. The package internally uses linear programming to compute equilibria and security strategies. Conventional real-valued matrix games are included as a special case and can be handled by the package too.

1. Introduction

During the last decades game theory evolved as a useful tool to analyse a variety of problems and is applied in many fields including economics, finance, operations research, conflict analysis, and risk management. With its increasing popularity and accordingly frequent use, many expansions of the classical setting introduced by Nash (1950) have been developed. In particular, strong assumptions such as full rationality of players have been relaxed to bounded rationality of players (Matsushima 1997) and the fact that players may, in reality, make mistakes can be modeled by trembling hand equilibria (Selten 1975). However, non-heuristic methods to express uncertainties about the consequences of actions (Rass 2018) beyond considering higher moments (besides the plain expected payoff), are relatively new (Rass, König, and Schauer 2015) and calls for software support. We therefore introduce the **HyRiM** package that supports zero-sum two player games with probability density-valued payoffs (not only mixing strategies) in R (R Development Core Team 2016). It is available from the

Comprehensive R Network (CRAN) (Rass and König 2019). Before illustrating its use with several examples, we give a short overview on the ideas behind games whose rewards are whole distributions rather than numbers, in the remainder of this section.

The basic issue is perhaps best illustrated with a simple decision problem between two random variables X and Y: given only their expectations, according to the popular quantitative risk management definition risk = impact × likelihood (cf., e.g., the ISO 31k Standard International Standards Organisation (ISO) (2018)), finding $\mathsf{E}(X) < \mathsf{E}(Y)$ would make action X preferable. But if X measures the loss of some action, then the expectation says nothing about the variation around it. Thus, action X may have decent chances to cause much more damage than the alternative action Y. Although Y will cause larger expected loss, its choice is nonetheless more reliable, since the overall possible damage is still lower than what action X can result in. Figure 1 depicts the problem (where the distributions are slightly ill-scaled for the sake of illustration only).

Many situations in risk management thus call for a more complex decision making, say, based on a stochastic tail order. That is what the package provides at the core, and what we discuss in the following sections, up to the point where whole games, as non-cooperative decision problems between multiple rationally acting entities, over stochastic orders can be defined. In security risk management, those entities are often two, being the defender against the attacker (physically decomposing into complex structures of interacting people and technical systems, but abstractly being here representable by two players).

1.1. Ordering Between Random Variables

While the space of probability distributions cannot be ordered in general, there exists a subspace on which a total order can be constructed based on a representation of a random variable by the sequence of its moments that can, in turn, be represented by a hyperreal number (Rass, König, and Schauer 2016). The restrictions imposed by this subspace are mild, defining a set $\mathcal{F}$ of loss distributions. We stress that this term differs from the more general one in actuarial science Klugman, Panjer, and Willmot (1998), and is tailored to the specific domain of security risk management, where our application originates.

Definition 1 (Loss Distribution) *Let X be a random variable that satisfies the following conditions:*

- *X has a known distribution F with compact support (note that this implies that X is upper-bounded).*

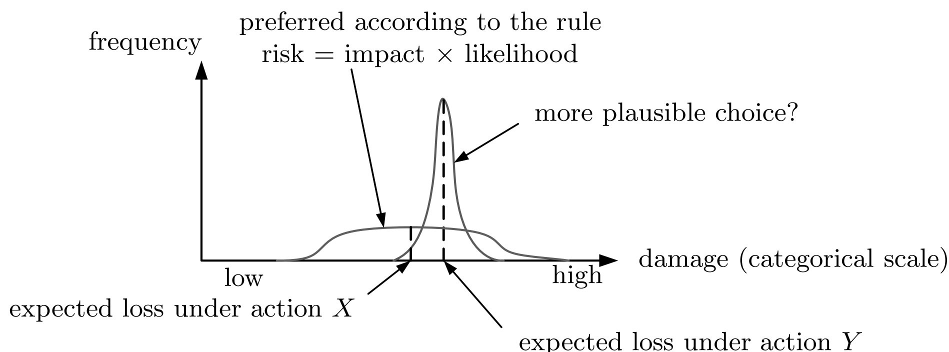


Figure 1: Example of risk evaluation issue

- $X \geq 1$ *(w.l.o.g., since as X is bounded, we can shift it into the region $[1, \infty)$).*
- *The probability measure induced by F is either discrete or continuous and has a density function f. For continuous random variables, the density function is assumed to be continuous.*

Then we think of X as a loss and call the distribution F of X a loss distribution*. The set of all such distributions is hereafter denoted as $\mathcal{F}$.*

Requirements 1 and 2 guarantee that all moments of X exists while Requirement 3 has purely technical reasons. On this set of probability distributions one can define a total ordering, via an embedding of distributions into the field ${}^*\mathbb{R}$ of hyperreal numbers, which is totally ordered. The embedding is one-to-one by replacing a distribution by its sequence of moments.

Definition 2 (Preference Relation) *We prefer a random variable X to a random variable Y if the sequence of moments of X diverges slower than the respective sequence for Y. The resulting order is denoted as $X \preceq Y \; :\iff \; \boldsymbol{x} = (\mathsf{E}(X^n))_{n\in\mathbb{N}} \preceq (\mathsf{E}(Y^n))_{n\in\mathbb{N}} = \boldsymbol{y}$ on the ordered field of hyperreal numbers $({}^*\mathbb{R}, \preceq)$.*

Details on the construction can be found in (Rass *et al.* 2015, 2016), where it is explicitly shown that the induced ordering on the random variables is, in fact, independent of the instance of ${}^*\mathbb{R}$ (i.e., the same in every ultraproduct defining the hyperreals).

It must be noted that the embedding into the hyperreal space is a purely technical move, and conveniently equips us with arithmetic, topologies and the totality of the ordering that comes for free (i.e., without needing any further proof). Nonetheless, we will later translate the rather abstract condition of Definition 2 into the more "handy" form (1), which is better suited to decide the order and establishes connections to other stochastic orders.

Why not the Usual Stochastic Order?

Let X, Y be random variables with densities f_X, f_Y, let $\Omega \subset \mathbb{R}$ be the union of the respective supports, and let $a := \max \Omega$. In case that $f_X(a) \neq f_Y(a)$, Definition 2 culminates in calling two random variables ordered as $X \preceq Y$ if and only if there is a value t_0 for which

$$\Pr(X > t) \leq \Pr(Y > t) \quad \text{for all } t \geq t_0, \tag{1}$$

i.e., the tails of X are lighter than those of Y. This is weaker than the usual (standard) stochastic order (Shaked and Shanthikumar 2006), defining $X \leq_{st} Y$ if and only if

$$\Pr(X > t) \leq \Pr(Y > t) \quad \text{for all } t. \tag{2}$$

Obviously, (2) defines only a partial ordering (we could just have X and Y with densities that oscillate around one another), whereas (1) is provably total (under the regularity conditions of Definition 1). Also, for risk purposes, the parameter t_0 has a natural interpretation as implying that low risks are usually not subject of extra risk management precautions, since they are up to the usual business continuity management. Risk management, on the other hand, is concerned with extreme events, such as about which (1) is explicit. The order used by the package does indeed bear relation to other stochastic orders; a more comprehensive discussion of this is given by Rass *et al.* (2016).

Special Case (Representation of Deterministic Values)

The case when losses are crisp is naturally included in the hyperreal construction: a deterministic value $X = a$, has a corresponding hyperreal representation as $\mathsf{E}(X^k) = \mathsf{E}(a^k) = a^k$ for all $k \in \mathbb{N}, a \geq 1$. That naturally enables a comparison of a value to a random variable, in the sense of Definition 1. Moreover, this comparison is practically meaningful, as it comes up $a \preceq X$ if and only if X has positive likelihood to take on any value larger than a (likewise, the comparison of two values $a, b \in \mathbb{R}$ comes up in $({}^*\mathbb{R}, \preceq)$ as it would in $(\mathbb{R}, \leq)$).

Since the package deals only with nondegenerate distributions, comparing values to distributions will not practically work in the way just described. However, Section 4.1 shows how to represent values $a, b \in \mathbb{R}$ as a Bernoulli distributions, to retain the relation $a \leq b$ in $\mathbb{R}$ in the same way using the Bernoulli representatives. The so-constructed distributions are here just technical vehicles, and do not provide any information about the original payoff (in particular, moments of these artificial distributions may have no practical use).

Deciding the Order of Discrete Random Variables

For discrete random variables, the $\preceq$-comparison of distributions boils down to a humble lexicographic ordering, as Lemma 1 makes precise (see (Rass *et al.* 2016) for a proof). This holds for ordered categorical variables (e.g., loss magnitudes ranging from “very low” to “very high”), so that they can equivalently be described by (integer) ranks.

Definition 3 (Lexicographic Ordering) *For two real-valued vectors $\boldsymbol{x} = (x_1, x_2, \ldots)$ and $\boldsymbol{y} = (y_1, y_2, \ldots)$ of not necessarily the same length, we define $\boldsymbol{x} <_{lex} \boldsymbol{y}$ if and only if there is an index i_0 such that $x_{i_0} < y_{i_0}$ and $x_i = y_i$ whenever $i < i_0$.*

Lemma 1 *Let F_1, F_2 be two random variables with support $\Omega = \{z_1 > z_2 > \ldots > z_n\}$, and let $\boldsymbol{f}_1, \boldsymbol{f}_2$ be the respective (empirical) density functions. Then $F_1 \preceq F_2 \iff \boldsymbol{f}_1 <_{lex} \boldsymbol{f}_2$, where $\boldsymbol{f}_i = (\Pr(F_i = z_1), \Pr(F_i = z_2), \ldots, \Pr(F_i = z_n)) \in \mathbb{R}^n$.*

So deciding $\preceq$ for discrete random variables can be done efficiently by deciding the lexicographic order between their empirical distributions, from right to left starting with the highest ranking category.

Deciding the Order of Continuous Random Variables

Let two continuous random variables $X \sim F, Y \sim G$ have smooth densities $f, g \in C^\infty([1, a])$ for some $a > 1$. Then the decision of $\preceq$ is possible using a sequence of derivatives of the densities (Rass *et al.* 2016):

Lemma 2 (Derivative Criterion) *Let $f, g \in C^\infty([1, a])$ for a real value $a > 1$ be probability density functions. If*

$$((-1)^k \cdot f^{(k)}(a))_{k \in \mathbb{N}} <_{lex} ((-1)^k \cdot g^{(k)}(a))_{k \in \mathbb{N}},$$

then $f \preceq g$.

The implemented functions make use of this criterion when working with smooth payoffs, here resulting from a kernel density estimation.

Remark 1 *Note that the densities of interest for us are all Lebesgue-integrable, so they have arbitrarily accurate approximations within C^∞ (by convolution). Thus, the hypothesis of Lemma 2 is indeed not too restrictive. The package internally uses a kernel density estimate based on Gaussian kernels, which therefore puts all distributions that the package uses into C^∞.*

1.2. Application in Risk- and Security Games

In the following the functionality of the **HyRiM** package is illustrated by applying it to scenarios from risk management which is the original field of application of (multi-goal) game-theoretic models Rass and Schauer (2018). In that setting, player 1 is the defender who tries to minimize the loss (payoff) he suffers due to an attack by player 2 for $d \geq 1$ security goals.

Hereafter, we consider games as non-cooperative competitions between two entities. To ease the description, we will let the utilities be all real values with the natural ordering $\leq$. The package later practically replaces those by density functions and a stochastic order $\preceq$ on them. Besides that, the theory of games itself remains unchanged. A general n-person game is a triple $\Gamma = (N, \mathcal{S}, H)$, composed from

- a set N of $n = |N|$ players; typically called by numbers (player 1, player 2, ...).
- a family $\mathcal{S} = \{PS_1, \ldots, PS_n\}$ of *pure strategies* (actions) that each player has. Each set PS_i is discrete and can take any individual number of elements, also of any type. Typically, an element $a_i \in PS_1$ is a more or less detailed (up to including textual) description of what the corresponding player does within each game iteration, and based on what the other players do. Thus, we implicitly include multi-stage games in which players can take alternating actions (i.e., games expressed in extensive form), and w.l.o.g., resort to the game being in *normal form*.

  To avoid complications with the existence of equilibria, we will hereafter adopt the symbol

  $$S(X) := \left\{ (p_1, \ldots, p_{|X|}) : p_i \geq 0\, \forall i, \text{ and } \sum_i p_i = 1 \right\},$$

  to mean the simplex spanned over the set X. In the game theoretic realm, this corresponds to the set of *randomized* actions that a player may take (say, acting differently at random upon every repetition of the game). Technically, the switch from PS_i to $S_i := S(PS_i)$ convexifies the action set, and thereby assures the existence of equilibria as proven by Nash (1950) (together with the continuity of the payoffs, which we describe next). Note that $X \subseteq S(X)$ always holds, by using degenerate distributions to represent each single action in X.
- A family H of payoff functions $u_1, \ldots, u_n$, each with domain $PS_1 \times PS_2 \times \cdots \times PS_n$. Each function u_i maps into $\mathbb{R}^d$ with $d \geq 1$. If $d > 1$, we write $\boldsymbol{u}_i$ and call Γ a *multi-objective* (synonymously *multi-criteria* or *multi-goal*) game. The real-valued description of the utility function is what the package mainly replaces by letting u map into a subset of the space of probability distributions, as Definition 1 describes.

The common object of interest given a game Γ is an *equilibrium*. This is a (perhaps mixed) strategy profile $(\boldsymbol{x}_1^*, \ldots, \boldsymbol{x}_n^*) \in S_1 \times S_2 \times \ldots \times S_n$, that, for all players $i \in N$ satisfies

$$u_i(\boldsymbol{x}_1^*, \ldots, \boldsymbol{x}_n^*) \leq u_i(\boldsymbol{x}_1^*, \ldots, \boldsymbol{x}_{i-1}^*, \boldsymbol{x}, \boldsymbol{x}_{i+1}^* \boldsymbol{x}_n^*) \quad \text{for all } \boldsymbol{x} \in S(PS_i), \tag{3}$$

where we assume that all players are minimizing (w.l.o.g., because otherwise, we can just sign-change the definition of the payoff for a maximizing player to make it minimizing). The package, as being designed for game theory in general but for risk management and security applications in particular, will generally assume *a minimizing* player 1 (defender) as we think of the payoff as a *loss*.

Equilibria in multiple goals are defined like regular Nash equilibria (3), but replace the $\leq$-relation by a vector-version thereof: let $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^n$ be given, then we write $\boldsymbol{a} < \boldsymbol{b}$ if $a_i \leq b_i$ for all coordinates $i = 1, 2, \ldots, n$. The complement of $<$ is denoted as $\geq_1$, i.e., we write $\boldsymbol{a} \geq_1 \boldsymbol{b}$ if and only if an index j exists for which $a_j \geq b_j$, no matter what the other coordinates do. A *Pareto-Nash* equilibrium is then defined exactly as (3), only treating $\leq$ as to hold per element (and taking $\geq_1$ to denote the complement relation that automatically holds upon violation of the equilibrium condition).

This concept will later be shown as essential to compute and characterize the notion that alternatively to an equilibrium is of central importance in security games. In such a competition, the defender acts as player 1, and seeks to achieve a "best defense" against an arbitrarily acting adversary (player 2), that is in particular *not obliged or otherwise motivated* to follow an equilibrium strategy together with the defender. That is, the incentive (i.e., utility structure) for the adversary is typically *unknown*, but as long as the defending player 1 knows the (full) action space for the other player, there is the notion of a security strategy in such security games. Intuitively, a *security strategy* is a behavior that may not optimize the revenues for a player, but guarantees a *minimum performance level*, here expressed as a worst-case loss under *any* behavior of the attacker within its action space PS_2. Definition 4 makes this rigorous, and at the same time, is the primary result object that the package computes.

Definition 4 (Multi-Goal Security Strategy with Assurance (MGSS)) *A strategy $\boldsymbol{p}^* \in S_1$ in a two-person multi-objective game with continuous payoff $\boldsymbol{u}_1 : S_1 \times S_2 \to \mathcal{F}^d$, with $\mathcal{F}$ as in Def. 1, for a (loss-)minimizing player 1, is called a* Multi-Goal Security Strategy *(MGSS) with* Assurance *$\boldsymbol{v} = (V_1, \ldots, V_d) \in \mathcal{F}^d$ if two criteria are met:*

Assurance: *The values in $\boldsymbol{v}$ are the component-wise guaranteed maximal losses for player 1, i.e., for all components i, we have*

$$u_1^{(i)}(\boldsymbol{p}^*, \boldsymbol{q}) \preceq V_i \qquad \forall \boldsymbol{q} \in S_2, \tag{4}$$

with equality being achieved by at least one choice $\boldsymbol{q}_i \in S_2$.

Efficiency: *At least one assurance becomes void if player 1 deviates from $\boldsymbol{p}^*$ by playing $\boldsymbol{p} \neq \boldsymbol{p}^*$. In that case, some $\boldsymbol{q}_{\boldsymbol{p}} \in S_2$ exists (that depends on $\boldsymbol{p}$) such that*

$$\boldsymbol{v} \preceq_1 \boldsymbol{u}_1(\boldsymbol{p}, \boldsymbol{q}_{\boldsymbol{p}}). \tag{5}$$

In the following we show in detail how security games can be set up and solved for MGSS in practice with help of the **HyRiM** package (Rass and König 2019) in R. For all the upcoming computations we assume that the package is installed and loaded by

```
library(HyRiM)
```

to have all functions available defined in there. We will show in detail how to construct loss distributions (both from continuous and discrete data) in Section 2, how to work with them in Section 3, how to construct games in Section 4, how to compute and understand equilibria in Section 5, how to customize plots in Section 6 and give general remarks on the implementation in Section 7. Concluding remarks are given in Section 8.

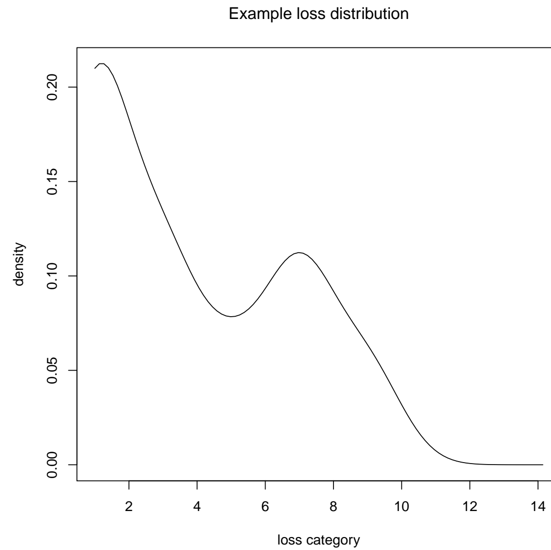


Figure 2: Example plot of loss distribution

2. Construction of Loss Distributions

For the initial demonstration let us create a bunch of arbitrary artificial observations,

```
dat <- c(rep(1,40), rep(3,20), rep(5,10), rep(7,20),
         rep(9,10));
```

from which a loss distribution can be constructed and plotted directly

```
ld <- lossDistribution(dat)
plot(ld,
     main="Example loss distribution",
     xlab="loss category",
     ylab="density")
```

as shown in Figure 2. The plotting function takes the usual additional parameters like titles, axes labels, axes limits, etc. and passes them onwards to the underlying (R-internal) plot routines.

The `lossDistribution()` function has some parameters to describe the input in more detail and customize the fitting. The example constructed above uses the default parameterizations, but the function can work with continuous and discrete data, internally using a kernel density approximation in both cases. We describe the respective applications separately.

2.1. Continuous Data

When observations yields samples from a continuous loss variable, the package uses loss distributions being Gaussian kernel density estimates. The construction allows for external supplies of bandwidth values but is bound to using the Gaussian kernel, since this choice allows a “closed form” computation of derivatives to decide the stochastic order using Lemma 2.

The remaining parameters of `lossDistribution()` apply only for categorical losses (discussed in Section 2.2) and are hence ignored for continuous data. The main challenge in the construction of accurate loss models is thus the proper choice of the bandwidth, as for any other kernel density estimator. Support by other specialized packages on kernel bandwidth choice is thus recommended.

Version 1.0.0.0 of the package does not support the construction of a loss distribution from a specific family, say, a Fréchet, Weibull, Gumbel, stable or other typical extreme value or general loss distribution. In such cases, a workaround is sampling random values from the distribution of interest and reconstructing the loss distribution from the so-obtained sample data. This method obtains an approximate loss distribution of the sought shape. Exact constructions are, however, possible for discrete loss distributions, as we will exemplify later.

Suppose we want to construct a truncated Weibull-shaped loss distribution, then we proceed as follows:

```
# sample from the desired distribution, and shift the
# values into the admissible range [1,inf)
d <- 1 + rweibull(n = 1000, scale = 1, shape = 5)
# reconstruct the loss with the sought shape. The bw
# parameter can replace the default bandwidth choice
# (bw.nrd0) by any more sophisticated computation
ld <- lossDistribution(dat = d, bw = bw.nrd(d))
```

Figure 3 shows a picture that overlays the standard kernel density estimate (obtained from `density(d)`), the true Weibull density, and the loss distribution constructed using the package.

2.2. Discrete Data

If the data is discrete, `lossDistribution()` accepts either observations (as in the continuous case; see Section 2.1), but also a pointwise defined probability mass function or the cumulative distribution function. To construct a discrete loss distribution, just add the flag `discrete = TRUE` when calling `lossDistribution()`.

Common to all constructions of loss distributions from categories is the need to specify how many categories there are. These are always named consecutively and numerically from 1 to a user-defined maximum number, and told by the parameter `supp`. For example, to specify a loss distribution on a simple 5 category scale "negligible" $<$ "minor" $<$ "medium" $<$ "large" $<$ "extreme", used in security risk management (Gouglidis, König, Green, Rossegger, and Hutchison 2018; König, Gouglidis, Green, and Solar 2018), we would use the parameter `supp = c(1, 5)` in all calls to `lossDistribution()`. These two parameters (`discrete` and `supp`) will thus appear throughout the upcoming examples.

Constructing Loss Distributions "by Name"

As of version 1.0.0.0, the package *does not* support direct specification of a loss distribution "by name", e.g., an $(a, b, 0)$-family, but this can be accomplished by evaluating the density and using the so-obtained data to construct it. For the $(a, b, 0)$-family, which is popular in actuarial science (Klugman *et al.* 1998), loss distributions are known to either be binomial, Poisson, negative binomial or Panjer, all of which admit the respective functions in R to compute density values that can be fed as data into `lossDistribution()`. For example, let us construct a Poisson distribution with rate

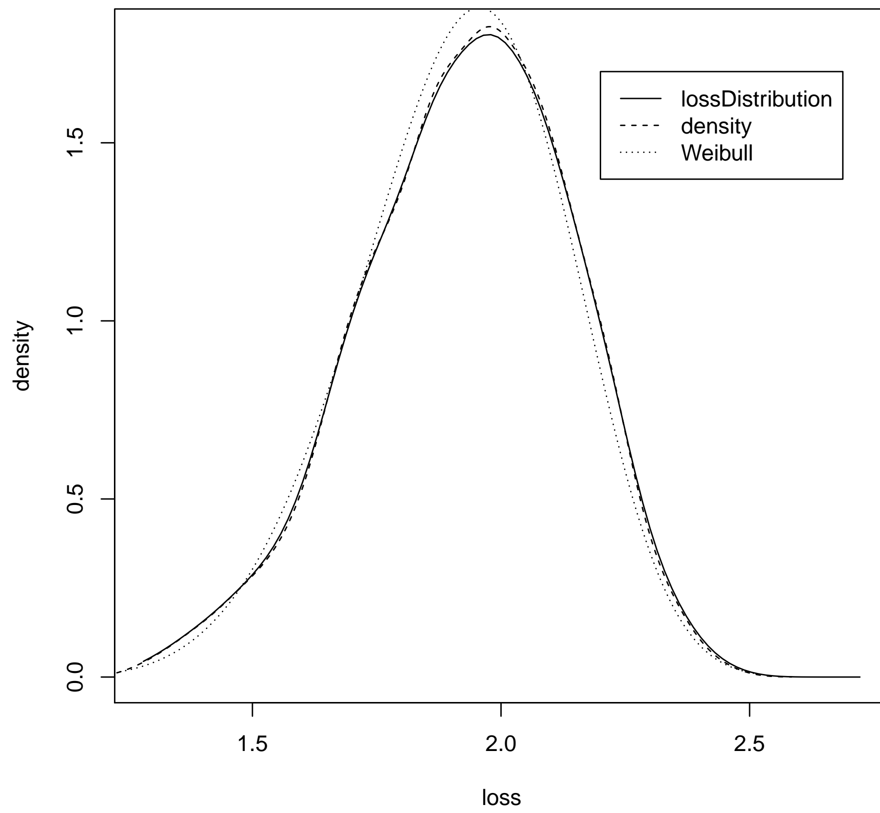


Figure 3: Example of Weibull-shaped loss distribution construction

parameter $\lambda = 3$ on the loss scale $1, 2, \ldots, 10$ "directly" using `dpois()` for the density. To this end, we tell the function that it is a discrete distribution (`discrete = TRUE`), the data describes the probability density function (`dataType = "pdf"`) and the support must be finite (the interval $[1, 10]$).

```
ld <- lossDistribution(dat = dpois(1:10, lambda=3),
                       discrete = TRUE,
                       dataType = "pdf",
                       supp = c(1,10))

## Warning in lossDistribution(dat = dpois(1:10, lambda = 3), discrete
= TRUE, :  renormalizing probability mass function
```

The warning issued by the function is a result from the fact that we truncated the distribution to the range $[1, 10]$ via specifying the support to be that range (`supp = c(1, 10)`), but did not ourselves truncate the mass function beforehand. Internally, `lossDistribution()` renormalizes the density to unit sum, which, essentially, is exactly what the (proper) truncation of the distribution would have done. So, the warning above informs us about this step that should have been done before calling `lossDistribution()`.

Like in the continuous case, `plot()` recognizes the type of the distribution and will output a bar plot. The same result would have been obtained by computing the cumulative distribution function and instructing `lossDistribution()` to use this kind of data, yielding the equivalent code sequence

```
ld <- lossDistribution(dat = ppois(1:10, lambda=3),
                       discrete = TRUE,
                       dataType = "cdf",
                       supp = c(1,10))
```

that differs from the above bit only in using `ppois()` for the distribution, and specifying the data type as `"cdf"` for *cumulative distribution function*. The plot, shown in Figure 4 from the latter construction, however, is identical to the output of the previous call using `dpois()`.

Using Sparse Empirical Data: Kernel Density Estimation and Smoothing

Upon raw observations (the default `dataType = "raw"`), `lossDistribution()` internally constructs a histogram using the provided support (parameter `supp`) as bins, i.e., categories. It may well happen that some of these bins remain empty, which typically leads to an error when the loss is used in a *game* since the theory (sketched in Section 1) requires the density to be nowhere zero (equivalently, the support must be a connected set). The problem, however, does not show up during the construction of the loss distribution, since as an object for themselves, zero likelihood categories are not forbidden per se. Therefore, the default behavior of `lossDistribution()` is to take the data *as it is* and construct the loss distribution from it. Problems may, however, arise when these distributions go into a game model later on. A small example creating and illustrating the problem is the following:

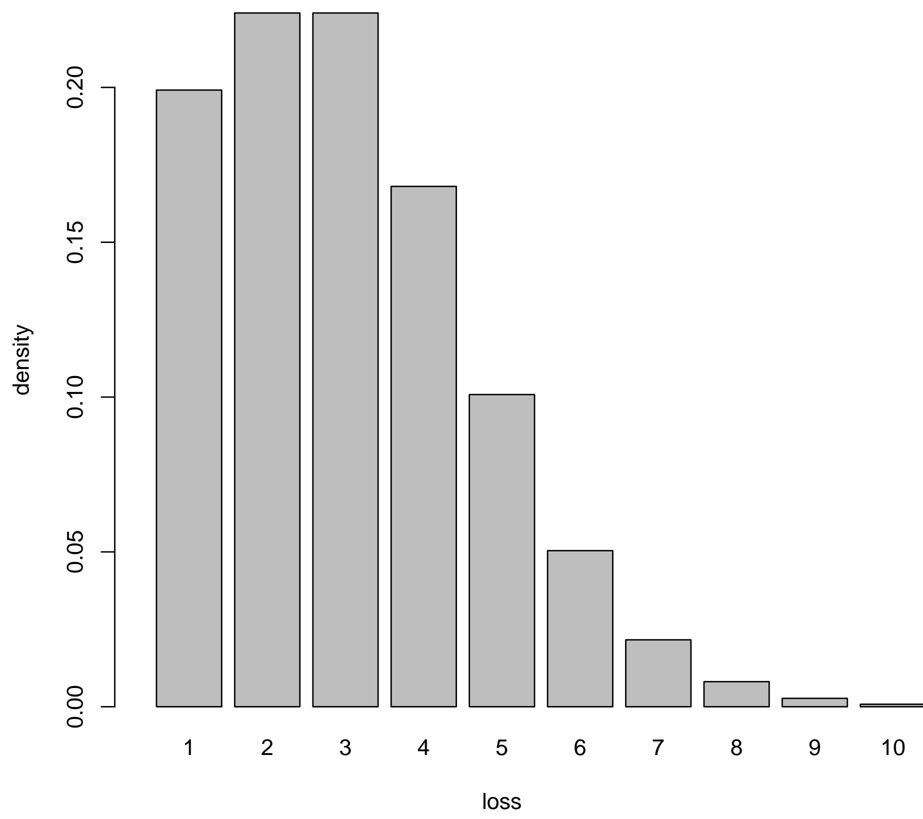


Figure 4: Poisson loss distribution example

```
# create a distribution with gaps (no observation of x=4)
ld1 <- lossDistribution(dat=c(1,1,1,2,3,3,5),
                        discrete = TRUE, supp = c(1,5))
# another distribution whose support is "too wide"
# (no observation of category 5)
ld2 <- lossDistribution(dat=c(1,1,2,3,3,4,4,4),
                        discrete = TRUE, supp = c(1,5))
```

In anticipation of how loss distributions are building blocks in multi-objective zero-sum games (see Section 4.2), let us look into a minimal example of a game using only these two distributions.

```
# construct the game
G <- mosg(n=2,m=1,goals=1, losses = list(ld1,ld2))
try(mgss(G))  # compute an equilibrium

## categorical distributions with empty categories are not
allowed. Consider reorganizing the game by smoothing the
loss distributions.Error in mgss(G) :
```

To handle this kind of error, we have the parameter `smoothing` that takes one of three values:

`none`: This is the default behavior for reasons as described above, but which can be problematic if observations are scarce.

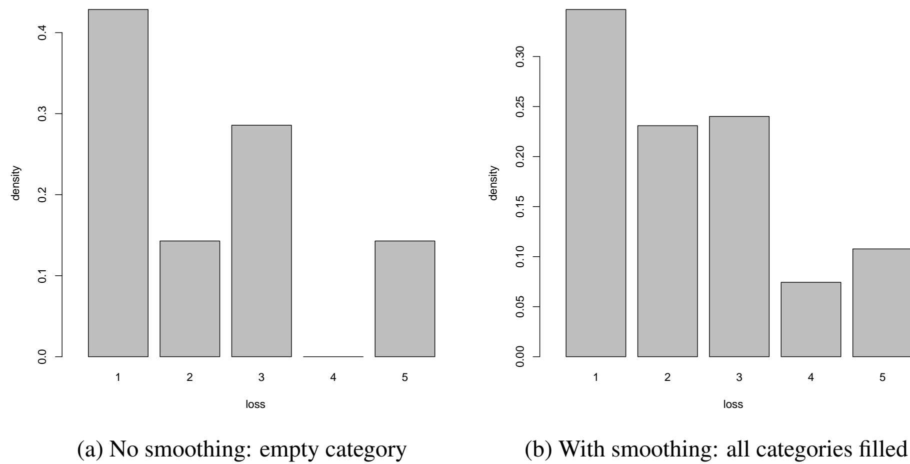
`always`: This applies a kernel-like smoothing to the data, which we describe in more detail below.

`ongaps`: This looks into the histogram and smoothes the data if and only if there are empty categories.

The theory of discrete kernel density smoothing is surprisingly sparse (Rajagopalan and Lall 1995; Li and Racine 2003; Kokonendji, Kiessé, and Zocchi 2007; Zougab, Adjabi, and Kokonendji 2012; Kiessé and Cuny 2013; Chu, Henderson, and Parmeter 2015) and only a few implementations seem to exist so far (Hayfield and Racine 2008; Wansouwé, Kokonendji, and Kolyang 2015; Wansouwé, Somé, and Kokonendji 2016). The package applies a simple heuristic to smooth categorical data: like as for continuous kernel density smoothing, it discretizes a Gaussian kernel of bandwidth h into a distribution that assigns $\Pr(X = n) = \Phi\left(n + \frac{1}{2}\right) - \Phi\left(n - \frac{1}{2}\right)$ for all $n \in \mathbb{N}$ and with Φ being the standardized cumulative Gaussian distribution function. For each category $c \in \mathbb{N}$, the function evaluates $\Pr(c \pm i)$ for $i = 1, 2, \ldots, 5$ and invokes `convolve()`, letting the convolution pad with zeroes outside the support by sending the parameter `type = "open"` to the function `convolve()`.

This method bears similar asymptotic features as conventional kernel density estimation, since letting $h \to 0$, the resulting discrete kernel degenerates into the assignment $\Pr(X = 0) \to 1$, so that the convolution's effect vanishes into returning the data ultimately unmodified when $h \to 0$. However, for large h, we get a local averaging to interpolate over empty categories by borrowing mass from nearby categories to fill the gaps. Practically, it admits the interpretation of assigning uncertainty, quantified by the parameter h, on the category assignment, and allowing a normally distributed error in it.

The parameter h, expressing the uncertainty, or interpreted as a parameter for the smoothing, is supplied by the parameter `bw()` to `lossDistribution()`. Giving an example:



(a) No smoothing: empty category (b) With smoothing: all categories filled

Figure 5: Example of Handling Empty Loss Categories

```
# same data as before for 'ld1', but now with smoothing
ld1smoothed <- lossDistribution(dat=c(1,1,1,2,3,3,5),
                       discrete = TRUE,
                       supp = c(1,5),
                       smoothing = "ongaps",
                       bw=0.5) # parameter "h" from above
```

Remark 2 *Recalling a well known result of Nadaraya (1965), one may wonder where the familiar condition* $h_n = c \cdot n^{-\alpha}$ *for* $c > 0$ *and* $0 < \alpha < \frac{1}{2}$ *has gone. The reason lies in the discreteness of the kernel; only the continuous case that Nadaraya's theorem speaks about requires this additional condition for convergence.*

3. Working with Loss Distributions

The density and cumulative distribution function from any loss distribution is available through the functions `density()` and `cdf()`. Both take the points at which the function needs evaluation, accepting vectors as inputs as well. Internally, the functions distinguish discrete from categorical distributions.

3.1. Summary Information and Statistics

Printing summaries uses the well known generic routines, such as:

```
ld  # same output as print(ld) or summary(ld)

##
```

```
## loss distribution for multiobjective security game
(MOSG)
##
## type: categorical
## loss range: 1 to 10
## mean: 3.04647960397752
## variance: 2.73121469797142
## quantiles:
## 10% 25% 50% 75% 90%
## [1,] 1 2 3 4 5
```

Moments are computed using either numerical integration (for continuous distributions) or by direct summation for categorical distributions. The `mean()` and `variance()` function rely on calls to `moment()` (either directly or using Steiner's theorem). The computation of quantiles uses a bisective search using the same method as `cdf()` internally in the function `quantile()`. Unlike `density()` and `cdf()`, the `quantile()` function takes only scalars (no vectors) as input.

3.2. Preference Relations

Deciding preferences between loss distributions is possible by invoking the `preference()` function. The function distinguishes discrete from continuous distributions but also accepts real-valued arguments, which are internally handled like degenerate distributions.

For non-degenerate distributions, the function applies a lexicographic order on the probability mass function (see Lemma 1). For continuous distributions, the function evaluates a sequence of points starting from the end of the support and extending towards zero, and searches for a region where one density function exceeds the other. Any such region (no matter how small) then determines the $\preceq$-order. The number of points into which the support is so discretized can be specified by the parameter `points`, which defaults to 512.

Remark 3 *The decision of preferences for the computation of equilibria works slightly different; it does not support comparisons involving numbers (as combining numbers with distributions is not supported by the package; see Section 4.4 for remarks on the reasons).*

Let us show a few examples, comparing distributions to other distributions and numbers.

```
preference(ld1, ld2, verbose = TRUE)

##
## loss distribution for multiobjective security game
(MOSG)
##
## type: categorical
## loss range: 1 to 4
## mean: 2.75
## variance: 1.4375
## quantiles:
```

```
## 10% 25% 50% 75% 90%
## [1,] 1 1 3 4 4

preference(ld1, 4) # compare to a fixed category (number)

## [1] 2

preference(ld, 16) # continuous distr. compared to number

## [1] 1
```

The parameter `verbose` defaults to `FALSE` and in that case returns either 1 or 2 (pointing towards the first or second parameter); returning 0 upon equality. If the output shall be verbose, the function returns the respectively preferred object.

4. Games

Games solvable by the package are always finite two-player competitions, with a finite number of goals that player 1 simultaneously *minimizes*. The game is, for each goal, zero-sum, making player 2 always be a maximizer.

4.1. Constructing Games with Stochastic Payoffs

Let us begin with the more conventional use case of the package, where a real-valued two player zero-sum matrix game ought to be solved. The function `lossDistribution()` needs at least two data points to compute a density due to the necessary bandwidth estimation, so we map real-valued payoffs to distribution-valued payoffs. The ordering between random variables introduced in Section 1.1 includes the order on the reals via a mapping of the real number $x \geq 1$ into a Bernoulli density on the set $\{1, 2\}$ with probability $\Pr(X = 2) \propto x$.

The definition of a game with normal numeric payoffs adopts this technique internally to construct Bernoulli (i.e., categorical) loss distributions from a list of payoffs. Since the basic pattern according to which the game is constructed from the list of payoffs applies in all cases, we illustrate the procedure by starting from a 3×4 real-valued payoff matrix and show how to solve this game using the package. For simplicity, we start with a single-objective game, introducing multiple goals later on.

Let the example payoff matrix be

$$\boldsymbol{A} = \begin{pmatrix} 4 & -1 & 4 & 2 \\ 3 & 4 & 1 & -2 \\ -2 & 1 & 3 & 2 \end{pmatrix},$$

which we assume has already been entered in R as variable `A`. The function `mosg()` (multi-objective security game) takes the dimension of the game via the parameters `n` (number of rows, actions for player 1) and `m` (number of columns, actions for player 2), and the number of `goals`.

The payoffs are supplied as a list through the parameter `losses`, which is a vectorized version of the payoff matrix (or matrices if there are more of them). If the payoffs are already available as a matrix,

then we can put the parameter `losses = as.vector(A)`, which serializes `A` column-by-column into the sought list. Since `as.vector` works column-by-column, we must instruct `mosg()` to read the list according to this pattern, which is done by the parameter `byrow = FALSE`.

Optionally, a list of textual descriptions for the defender's, attacker's strategies and goals can be supplied via the parameters `defensesDescr`, `attacksDescr` and `goalDescriptions`.

The construction of a game proceeds by calling

```
G <- mosg(n = 3, m = 4, goals = 1,
          losses = as.vector(A), byrow = FALSE)

##
##  Multiobjective security game
##
## Shape: 3 x 4
## Goals: 1
##
## Goals:
##   1
## Defense strategies:
##   1   2   3
##
## Attack strategies:
##   1
##    2
##    3
##    4
```

Remark 4 *Observe that the "direct" construction of games from numbers* allows *supplying negative values (thus bypassing the restriction imposed by the regularity assumptions given in Definition 1 in Section 1.1). The function internally shifts all data uniformly into the positive range before defining the Bernoulli payoffs from there. Figure 6 shows what the plot of the game generated by* `plot(G)` *looks like.*

The function `mgss()` runs runs through a sequence of linear programs to obtain an equilibrium/security strategy. We postpone the details of the respective function until Section 5.

```
mgss(G)

##
##  equilibrium for multiobjective security game (MOSG)
##
## optimal defense strategy:
##        prob.
## 1 0.2391304
## 2 0.5434783
```

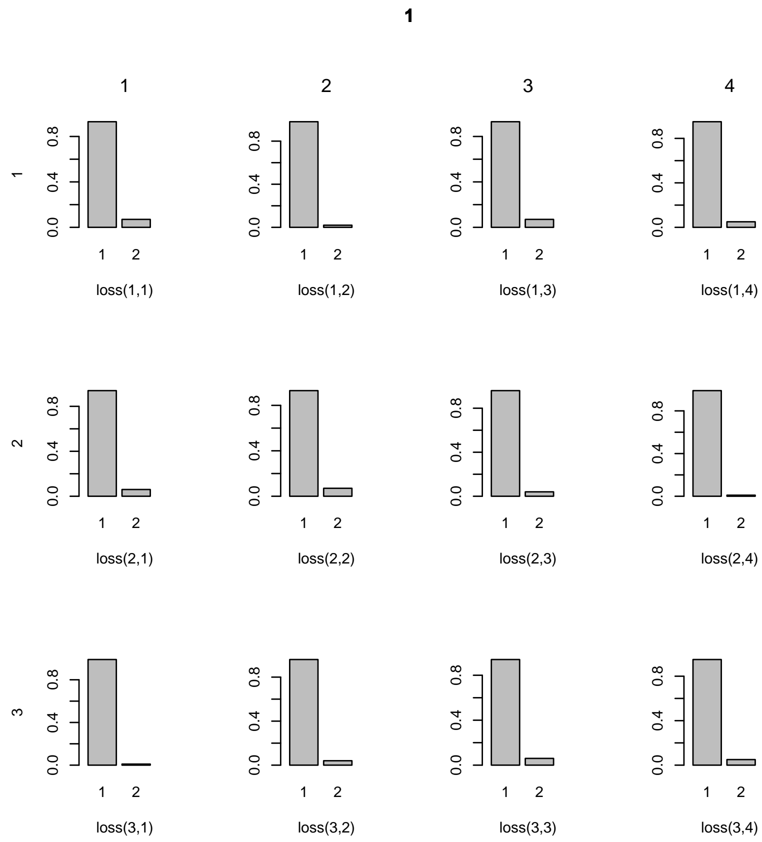


Figure 6: Example of numeric game represented by Bernoulli distributions

```
## 3 0.2173913
##
## worst case attack strategies per goal:
##              1
## 1 0.02173913
## 2 0.36956522
## 3 0.60869565
## 4 0.00000000
```

The construction of games using probability distributions as payoffs proceeds according to the same procedure, with the only difference being the list of losses being replaced by a list of objects constructed via `lossDistribution()` (as discussed in Section 2).

Remark 5 *Version 1.0.0.0 of the package does not support constructing games from equilibrium distributions. The reason lies in the internal representation of a loss distribution that differs between loss distributions obtained by* `mgss()`*, and those returned directly by* `lossDistribution()`*. The latter uses a representation as a mix of kernel densities, while the other is a pointwise defined curve. As of version 1.0.0.0, there is no way to define loss distributions pointwise, the respective use of equilibrium distributions in further game models is thus not supported (in this version).*

4.2. Multiple Goals

Multiple payoff structures need to be vectorized in all the same fashion (all row-by-row or all column-by-column), and go into the `losses` list consecutively, i.e., letting $vec(\boldsymbol{A}_i)$ denote the vectorized version of the payoff matrix $\boldsymbol{A}_i$ for $i = 1, 2, \ldots, d$, the list of losses supplied to `mosg()` takes the general structure

$$vec(\boldsymbol{A}_1), vec(\boldsymbol{A}_2), \ldots, vec(\boldsymbol{A}_d)$$

For instance, the matrices $\boldsymbol{A}_1 = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ and $\boldsymbol{A}_2 = \begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix}$ would vectorize into the list $(1, 2, 3, 4, 10, 20, 30, 40)$ row-wise per goal.

An example of such a game using continuous distributions and three security goals has three payoff structures, accordingly. Figure 7 shows a concrete example of a 2×2 game with two goals.

4.3. Computing Loss Distributions from Given Strategies

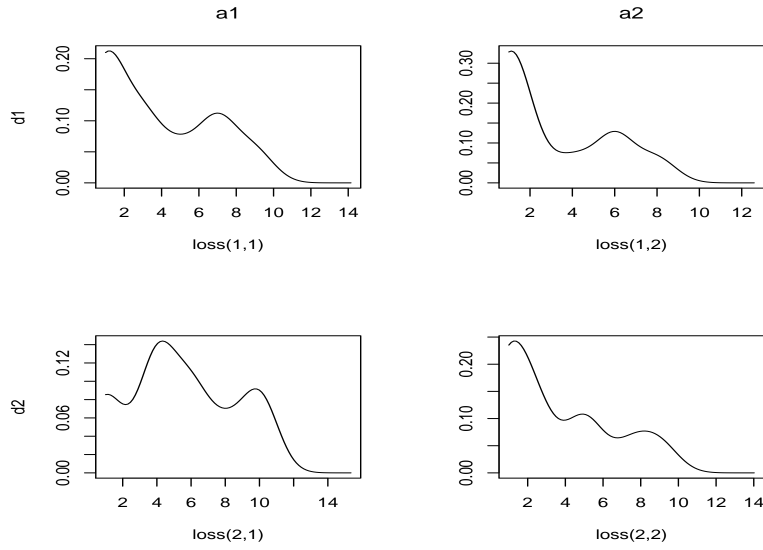
The loss that a game causes for player 1 is for arbitrary mixed strategies $\boldsymbol{x} \in S(PS_1), \boldsymbol{y} \in S(PS_2)$ given by

$$\boldsymbol{x}^T \cdot \boldsymbol{A}_i \cdot \boldsymbol{y}, \tag{6}$$

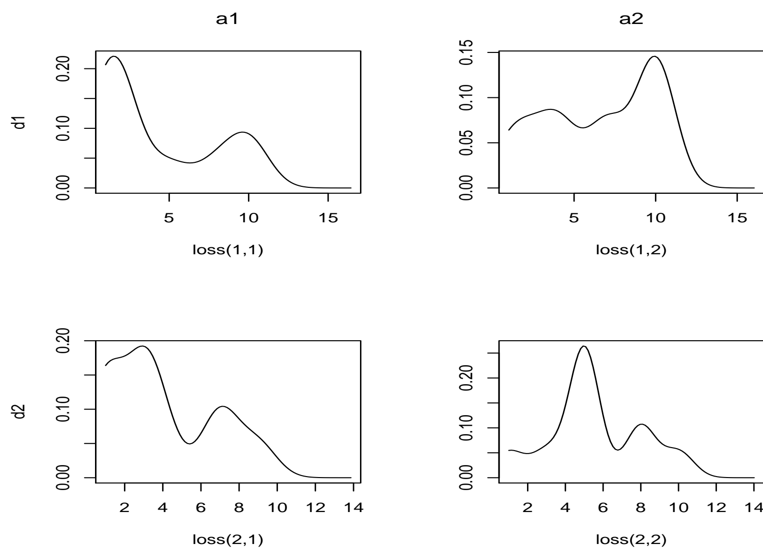
regarding the i-th goal, i.e., loss matrix, $\boldsymbol{A}_i$ in the game (for $i = 1, 2, \ldots, d$).

For a chosen mixed strategy, (6) can be evaluated as follows:

```
# evaluate the payoff distribution for x = (1/2, 1/2),
# and y = (1/3, 2/3) for the game G1 with
# continuous losses
```



(a) First goal g1



(b) Second goal g2

Figure 7: Example of a multicriteria game

```
my_ld <- lossDistribution.mosg(G1,
                               player1Strat = c(1/2, 1/2),
                               player2Strat = c(1/3, 2/3),
                               goal = 1,
                               points = 512)
print(my_ld)

##
##  loss distribution for multiobjective security game (MOSG)
##
## type: continuous
## loss range: 1 to 10
## mean: 4.68468179867139
## variance: 7.49774749625169
## quantiles:
##           10%      25%      50%      75%      90%
## [1,] 1.429291 2.168121 4.325287 6.770294 8.679718
```

This function computes (6) pointwise at the specified number of points (512 in the example) over the (common) support of all losses in the game, and for the goal, specified *by index* (not name) through the parameter `goal`. Figure 8 plots the result.

4.4. Mixing Numeric and Stochastic Payoffs

Payoff types, numeric and stochastic, cannot jointly occur in the same game; the function `mosg()` internally checks for a homogeneous type of all payoffs. The theory admits combining all these objects on the grounds of the common representation of a distribution by moment sequences. However, such a combination is not necessarily meaningful in practical applications. Consider the following combinations:

- Continuous + Categorical: this means doing arithmetic on incompatible scales (ordinal outcomes added to numeric rewards).
- Continuous + Numeric: numeric payoffs correspond to singular distributions (point masses). Though the stochastic order $\preceq$ covers these cases, the preference between a numeric and a stochastic payoff would be fixed by definition of the two payoffs. It is possible to trick the package into using a degenerate distribution if this is explicitly demanded; we describe how to do this in Section 7.1.
- Categorical + Numeric: this would create the same problems as in combination with the continuous + categorical combination.

For these practical reasons, the package does not support any such combinations. Comparisons between numeric and continuous, resp., categorical, distributions are, however, directly supported through the `preference()` function (see Section 3.2).

5. Equilibria

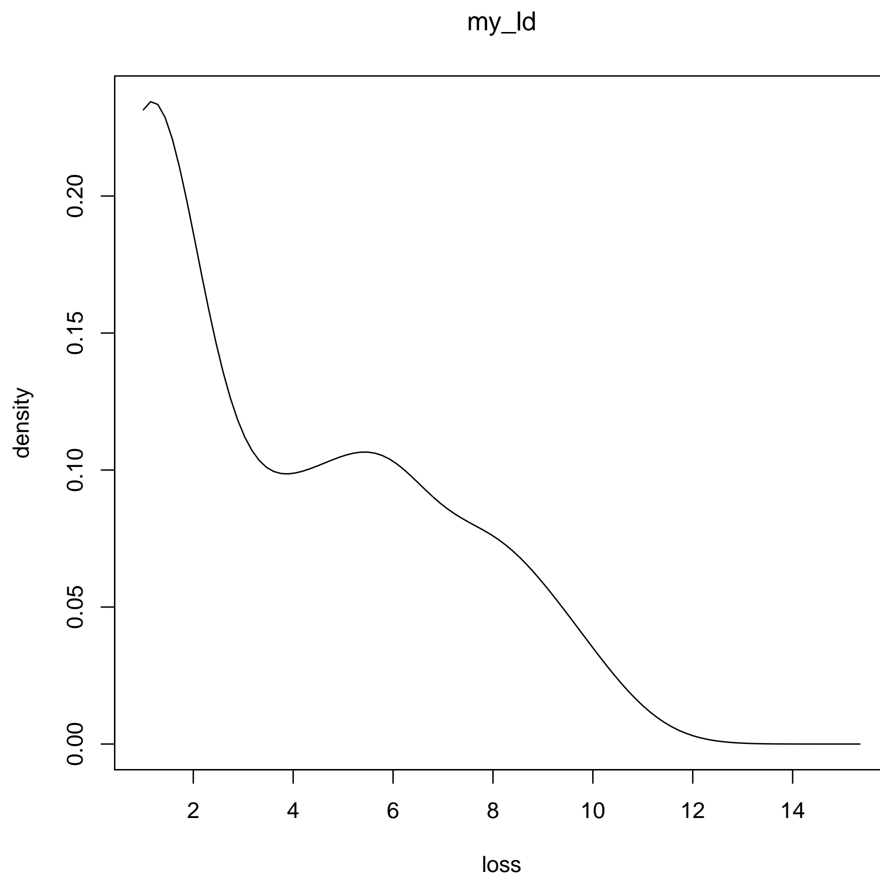


Figure 8: Example evaluation of equation (6)

A multi-objective security game constructed by `mosg()` can directly be fed into the `mgss()` function to compute an equilibrium object (of class '`mosg.equilibrium`'), according to Definition 4. In the sequel, we illustrate how the parameters of the `mosg()` can be adapted for the computation of an equilibrium. Before going into the syntax to do the computation, we use the next section to open up on some background.

5.1. Computation of MGSS

If the game has multivariate payoffs per player, we need to optimize several goals simultaneously to compute a MGSS by Definition 4. The trick is a transformation of the two-person game with $d \geq 1$ goals into a $(d+1)$-person game, in which the original defender enters as player 0, opposing one opponent per goal, i.e., a total of d attackers. The so-defined one-against-all competition is called an auxiliary game in this context.

Definition 5 (Auxiliary Game) *For a multi-objective two-player game Γ with payoff structure $\boldsymbol{F}_1 = (F_1^{(1)}, \ldots, F_1^{(d)})$ of $d \geq 1$ dimensions and arbitrary (unknown) payoff $\boldsymbol{F}_2$, $\Gamma = (\{1,2\}, \{S_1, S_2\}, \{\boldsymbol{F}_1, \boldsymbol{F}_2\})$, we define a $(d+1)$-player multi-objective game $\overline{\Gamma} = (N, S, H)$ through:*

- *$N = \{0, 1, \ldots, d\}$ is the set of players*
- *$S = \{S_1, S_2, \ldots, S_2\}$ is the strategy multiset containing d copies of S_2 (one for each opponent in N)*
- *the payoffs for all player*
  - *$\overline{\boldsymbol{F}}_0(s_0, \ldots, s_d) := (F_1^{(1)}(s_0, s_1), \ldots, F_1^{(d)}(s_0, s_d))$ for player 0*
  - *$\overline{F_i}(s_0, \ldots, s_d) := -F_1^{(i)}(s_0, s_i)$ for each opponent $i = 1, \ldots, d$*

The game $\overline{\Gamma}$ is called the auxiliary game *for Γ.*

The constructed auxiliary game allows computation of a Nash equilibrium by an algorithm due to Lozovanu, Solomon, and Zelikovsky (2005):

Theorem 3 *Let $\overline{\Gamma} = (\{0, \ldots, d\}, \{S_0, \ldots, S_d\}, \{\boldsymbol{F}_0, \ldots, \boldsymbol{F}_d\})$ be a $(d+1)$-player multi-objective game, where $S_0, \ldots, S_d$ are convex compact sets and $\boldsymbol{F}_0, \ldots, \boldsymbol{F}_d$ represent vector-valued continuous payoff functions (where the payoff for player i is composed from $r_i \geq 1$ values). Further assume that for every $i \in \{1, \ldots, d+1\}$ each component $F_i^{(k)}(s_0, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_{d+1})$, $k \in \{1, \ldots, r_i\}$, of the function $\boldsymbol{F}_i$ represents a concave function w.r.t. s_i on S_i for fixed $s_0, \ldots, s_{i-1}, s_{i+1}, \ldots, s_{d+1}$. Then the multi-objective game $\overline{\Gamma}$ has a Pareto-Nash equilibrium.*

Theorem 3, in connection with the next result, is the key to finding multi-objective security strategies in the sense of Definition 4 (see (Rass 2013) for a proof):

Theorem 4 *Let Γ be a two-player multi-objective game with $d \geq 1$ distribution-valued payoffs. The strategy $\boldsymbol{p}^*$ is a security strategy with assurance $\boldsymbol{v}$ (according to Definition 4) in the game Γ, if and only if, it is a Pareto-Nash equilibrium strategy for player 0 in the auxiliary $(d+1)$-player game $\overline{\Gamma}$ according to Definition 5.*

Theorem 4 allows computation of a security strategy that simultaneously optimizes $d \geq 1$ goals by translating the original game into a one-against-all game with just one goal for the defender (player 0 in the auxiliary game). The attacker, on the other hand, may need to deviate from his optimal strategy if these are different for different goals (represented by different opponents i) in which case the actual loss may be smaller than the predicted worst-case loss (i.e., the assurance).

Algorithmic Matters of Computing MGSS

Now, here comes the catch: our embedding of probability distributions into the hyperreal space equips us with well-defined arithmetic, but leaves us unable to *perform* all computations in the hyperreal field (especially divisions are troublesome in absence of an explicitly represented ultrafilter). Thus, we are bound to algorithms that use a minimum lot of arithmetic. Past versions 1.x.x of the package used Fictitious Play (FP), giving only approximate equilibria. This was replaced by linear programming in version 2.0.0. The method relies on the fact that the stochastic order is in some cases equivalent to a lexicographic ordering on properly constructed vectors. That is, we first play a game using the payoff structure using just the last coordinate, and obtain a saddle point value v_1 there. Then, moving to the second-last coordinate, we search for an equilibrium under the additional constraint to not worsen the performance v_1 on the last coordinate that we optimized already. This comes to a selection of optima among equilibrium strategies, and in a way "refines" the equilibrium found in the first round to one that also optimizes the payoffs in the second coordinate. Calling the optimum v_2, we move to the third-last coordinate, playing a game there as usual, but under the two constraints to preserve the payoff v_1 on the last and v_2 on the butlast coordinate, and so on. In this way, we run through a sequence of linear programs to give us a lex-order optimal solution, corresponding to a $\preceq$-optimum as desired. The algorithm is found in more detail in Rass (2015).

Let us wrap-up what we have: Theorem 4 equates the computation of MGSS to the computation of Pareto-Nash equilibria, and Theorem 3 assures the existence of the latter under conditions that our game models always satisfy (as being finite games with linear and hence continuous payoff functionals). Moreover, Lozovanu *et al.* (2005) give a constructive method to compute a Pareto-Nash equilibrium as we seek it: it starts with a scalarization of the multi-objective game into a single-objective game to get back a zero-sum game where linear programming can be used to find equilibria.

Definition 6 (Scalarized Game) *For a multi-objective two-player game Γ with payoff structure $\boldsymbol{F}_1 = (F_1^{(1)}, \ldots, F_1^{(d)})$ of $d \geq 1$ dimensions and arbitrary payoff $\boldsymbol{F}_2$, $\Gamma = (\{1,2\}, \{S_1, S_2\}, \{\boldsymbol{F}_1, \boldsymbol{F}_2\})$ and the corresponding $(d+1)$-player multi-objective game $\overline{\Gamma}$ (as defined in Definition 5), we define a* scalarized game $\overline{\Gamma}_{sc}$ *by first picking a set of values $\alpha_i > 0$ for all goals $i = 1, 2, \ldots, d$ and defining the game to use N and S from Definition 5, and the scalar payoffs f_i for all players given by:*

- $f_0(s_0, \ldots, s_d) = \alpha_1 \overline{\boldsymbol{u}_1} + \ldots + \alpha_d \overline{\boldsymbol{u}_d}$ *for player 0 where $\overline{\boldsymbol{u}}$ is player 0's payoff function in the auxiliary game*
- $f_i = \alpha_1 \cdot 0 + \ldots + \alpha_{i-1} \cdot 0 + \alpha_i \cdot (-u_1^{(i)}) + \alpha_{i+1} \cdot 0 + \ldots + \alpha_d \cdot 0$ *for each opponent $i = 1, \ldots, d$, where $u_1^{(i)}$ is the payoff from the original two-person multi-objective game Γ.*

The weights α_i appearing in Definition 6 are of particular use to prioritize goals (Section 5.2 will come back to this).

For numerical computation of an equilibrium, one more transformation due to Sela (1999) is needed to construct a zero-sum game. This so-called *reduced game* $\overline{\Gamma}_{scr}$ is a two-player game where the

second player's payoff is the sum of the payoffs of all opponents of player 0. The resulting game is zero-sum by construction and allows for computing equilibria by linear programming. Unfortunately, (counter)examples show (Rass 2015) that we need the additional hypothesis of all payoff densities to be strictly positive on the same interval (and hence the package enforces it). This condition is assured by using Gaussian kernels (whose support is the entire line $\mathbb{R}$) and truncating them all at the same point (called the "cutoff" point in the context of the package, and further discussed in Section 5.2).

5.2. Customizing the Equilibrium Computation

The parameters of `mgss()` allow customization of the equilibrium computation.

Goal Prioritization

When computing a MGSS, the weights α_i (as they appear in Definition 6) induce a degree of freedom which can be used to *prioritize* the different goals (from player 0's point of view). They can be understood as representing the different priorities of the corresponding goals, i.e., the most important goal gets the highest weight. There is no restriction on how the goals are relatively weighted, as long as all α-values are strictly positive. For example, if goal g_1 is half important as goal g_2, we may choose the weights to satisfy $2 \cdot \alpha_{g_1} = \alpha_{g_2}$.

Technically, the weights α_i enter the `mgss()` function to compute an MGSS through the argument `weights`. If missing, it defaults to a uniform weight assignment on all goals, assuming equal importance unless specified otherwise.

Cutting the Payoff Distributions

The `mgss()` function takes the following parameters:

`ord`**:** Based on Lemma 2, the package internally approximates the kernel at the tails using a Taylor series expansion of order `ord` at $x =$ `cutOff`. It defaults to 5, since the preference is (with probability 1) determined by the value of the density at $x =$ `cutOff` (again, by Lemma 2).

`cutOff`**:** This value serves two purposes:

1. The truncation of the loss distribution assures the existence of all moments. The truncation point defaults to the maximal data range supplied, but practically *should be fine-tuned*, since it crucially influences the stochastic orders among the payoffs.
2. It determines the risk region where the preference is decided. For example, within a risk management process (like ISO31000), one step in the management workflow is the specification of a threshold of maximal damage, beyond which no further distinction is made (losses above this value simply count as "extreme" and must be avoided in any case. This is exactly the value that needs to be supplied by the parameter `cutOff`.

`fbr`**:** if set to `TRUE`, instruct the function to additionally compute the best replies regarding each goal individually, assuming that defender plays `optimalDefense` as a leader, and the attacker per goal follows (follower's best reply). These replies are always pure strategies.

`points`**:** This is the number of points at which the resulting assured loss distributions are approximated by `lossDistribution.mosg`. The `mosg.equilibrium` object returned by `mgss()` contains a spline function to describe the resulting mix density.

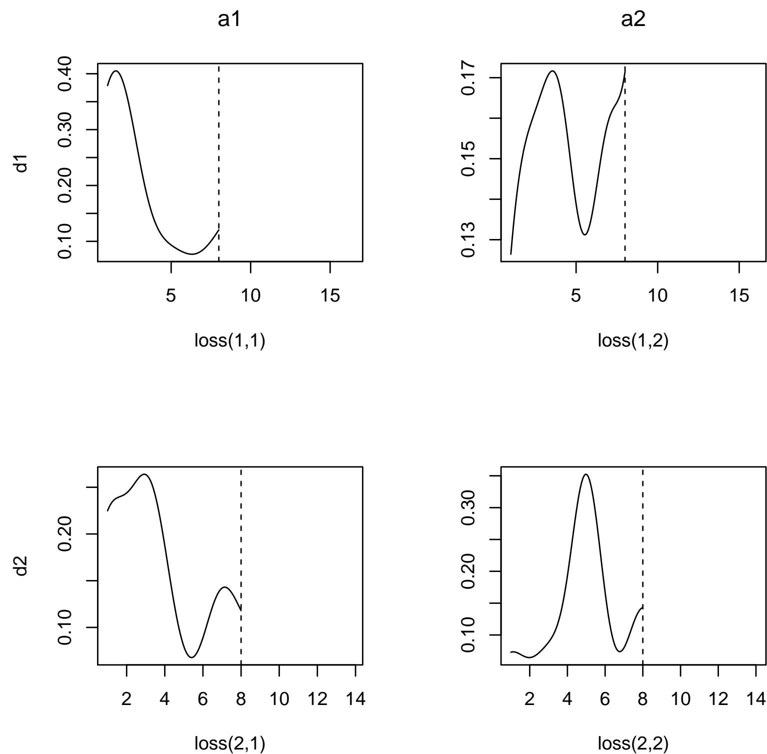


Figure 9: Plotting distributions truncated at x =`cutoff`

Remark 6 *This is the technical reason why equilibrium loss distributions cannot be used to construct further games, since these expect a kernel density sum representation of the loss distributions internally.*

For numeric stability, however, it is advisable to set `cutOff` not too far off the numeric range of the data, since the derivatives of the Gaussian kernels vanish relatively fast (thus creating unwanted roundoff errors). The default for `cutOff`, however, is the full numeric range of the data, so this value *should* be set explicitly.

The cutoff point has a substantial impact on the resulting equilibrium. For example, computing an equilibrium if the payoff distributions are truncated at losses ≤ 5 entirely changes the optima for both, the defender and the attacker, compared to the computation from before (by `mgss(G1)`):

```
mgss(G1, cutOff = 5)

## Loading required package:  polynom

##
```

```
##  equilibrium for multiobjective security game (MOSG)
##
## optimal defense strategy:
##            prob.
## d1  1.000000e+00
## d2 -2.020912e-16
##
## worst case attack strategies per goal:
##    g1            g2
## a1  0 -1.616416e-15
## a2  1  1.000000e+00
```

The so-modified game from Figure 7 is shown in Figure 10.

Special Cases in Security Risk Management

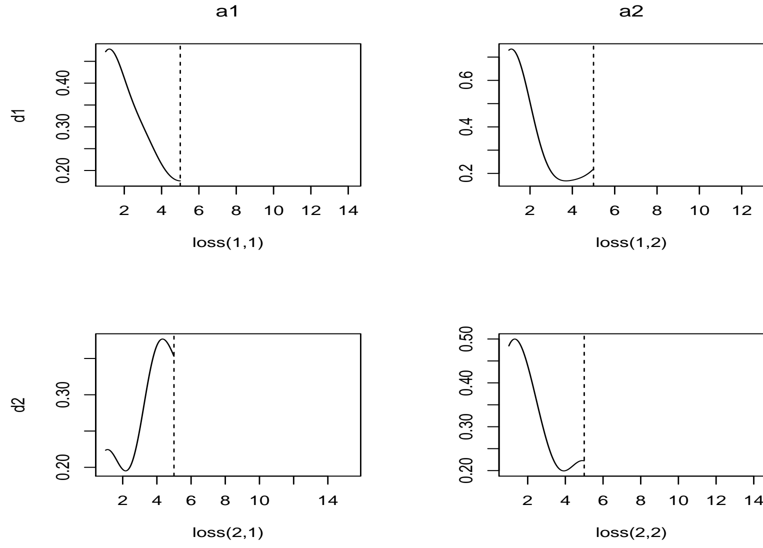
Two special cases of two-player zero-sum games are of interest despite their simple structure.

- **Evaluation of the status quo: A fixed defence**
  In the situation where the defender does not have several options to protect his system (that is, his only defence strategy is the current state) the attacker can directly choose the attack that maximizes the expected damage because the defenders action is deterministic. In this situation the payoff matrix has dimension $1 \times m$, i.e., it reduces to a row vector. The optimization turns is thus just a maximum computation over a finite set of random variables (that satisfy our assumptions of Definition 1). Practically, the defender may use this setting to find out which is the most severe attack in a known list of threats.

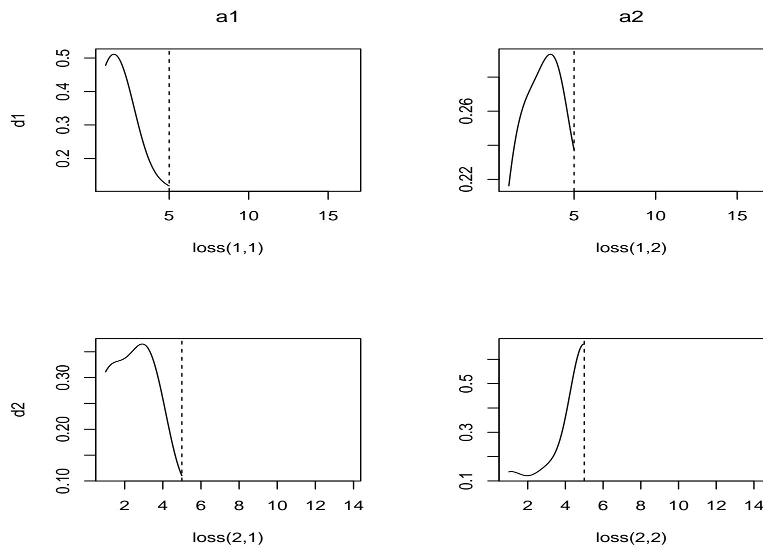
- **A single attack: Choosing the best among several defences**
  On the other hand, if we only consider an attacker with only one attack strategy the defender can directly minimize his expected damage by choosing his best defense, so the optimization turns into a minimization on a set of loss distributions.

Both cases can be treated by applying the `preference` function iteratively, i.e., by always comparing two distributions at a time. A direct application of this minimization of uncertain variables is classical risk management. If the (random) consequences of a risk are described by random variables as used here, the $\preceq$ ordering may be used during *risk evaluation*. Application to the concept of risk matrices is illustrated by Schauer, König, Latzenhofer, and Rass (2017).

Things are a bit more involved if several goals are of interest, e.g., if the impact is measured in terms of money and in terms of reputation. A canonical way to rank two variables is to define a game with multiple goals and corresponding degenerated payoff matrices (say, $1 \times m$ where the defender only has one choice). However, such a game does *not* directly yield the most severe attack since the equilibrium is computed through the $(m+1)$-player auxiliary game where each opponent acts independently of the other players. So the equilibrium returns a worst-case attack per goal, which does not necessarily coincide with the aim of a multi-objective risk optimization. On the other hand, the theory can be applied when the game is reduced to a single two-player $1 \times m$ game with scalarized payoffs. This game yields a pure equilibrium that indicates the worst case attack for player 1.



(a) First goal g1



(b) Second goal g2

Figure 10: Game with loss distributions truncated at cutoff point $x = 5$

5.3. Representing the Equilibrium

In order to really understand the computed equilibrium, it is helpful to represent it in several ways. The generic `summary()`, `print()` and `plot()` functions exist for the resulting equilibrium object and allow a deeper understanding of the found equilibrium.

Textual Representation

Let us take the game from Figure 7 (here called `G1`) towards a solution, that in addition includes the best replies if the defender were to play a fixed strategy, i.e., the attacker would be a follower. We get this auxiliary information by setting the flag `fbr`:

```
eq <- mgss(G1, fbr = TRUE)
print(eq)

##
##  equilibrium for multiobjective security game (MOSG)
##
## optimal defense strategy:
##    prob.
## d1     0
## d2     1
##
## worst case attack strategies per goal:
##                 g1 g2
## a1 1.000000e+00  0
## a2 3.459184e-15  1

## Warning in if (!is.na(x$br_to_optimalDefense)) {:  Bedingung hat Länge
> 1 und nur das erste Element wird benutzt

##
## best replies per goal, following the (fixed) optimal defense strategy:
##      1 2
## [1,] 1 2
```

The textual summary description of the equilibrium displays the optimal defense as a categorical distribution over the action space, labeled with the previously assigned descriptive texts (supplied as the parameter `defensesDescr` to `mosg()`). The example game in Figure 7 has a pure security strategy being `d2`, under *equal* goal priorities, which is the default setting for the α values in Definition 6. Figure 7 visually explains this result with the relatively large losses suffered regarding goal `g2` upon playing strategy `d1`, since the losses range up to ≈ 13, as opposed to losses bound below 12 under the alternative strategy `d1`. More specifically, disregarding goal `g2` entirely, strategy `g1` would be clearly preferable. We can confirm this intuition by excluding the second goal from the equilibrium calculation. To this end, we may assign a very small (yet necessarily positive) weight to the goal to be excluded, e.g., supply `weights=c(0.999,0.001)` upon calling `mgss()` to exclude the second goal.

The lowest part labeled with `best replies following the (fixed) optimal defense` gives the best replies of the attacker following the attacker. Formally, the attacker would make its choice then from the row-vector $((\boldsymbol{x})^*)^T \cdot \boldsymbol{A}$, always leading to a pure strategy best reply in this follower scenario. The function returns the *index* of the respective best response as a pointer into the list of strategies. If attack descriptions are available, a more descriptive result can be obtained by calling `G$attacksDescriptions[eq$br_to_optimalDefense]`.

The equilibrium object carries information about assurance as defined in Section 1.2, and these values are accessible as fields of the resulting object. If strategies and goals have been named, these names are useful for accessing the respective objects directly. Again, for the example game plotted in Figure 7:

```
# get the optimal defense
eq$optimalDefense   # this named field is always available

##    prob.
## d1     0
## d2     1

# get the worst-case attack strategies
eq$optimalAttacks   # this named field is always available

##              g1 g2
## a1 1.000000e+00  0
## a2 3.459184e-15  1

# get the assured loss for, say, the second goal "g2",
# either by accessing the second element in the list
# (assurances[[2]]) or by name:
eq$assurances$g2    # the "assurances" field always exists

##
##  loss distribution for multiobjective security game (MOSG)
##
## type: continuous
## loss range: 1 to 10
## mean: 5.77388201578312
## variance: 5.5565232808748
## quantiles:
##           10%      25%     50%      75%      90%
## [1,] 2.887177 4.286285 5.31076 7.611298 9.241669
```

Each object in the list accessible by `$assurances` is of class `mosg.lossdistribution()`, and carries the following fields:

`lossdistr`**:** A function taking a single value and returns the density function's value of the loss distribution. The inner form of that function depends on whether the object was constructed by `lossDistribution()` or `mgss()`:

- for `lossDistribution()`, it is a kernel density estimate (explicitly represented as a sum of Gaussian kernels).
- for `mgss()`, it is a spline, interpolated at the specified number of points by using `lossDistribution.mosg()` internally as described in Section 4.3. This number is the parameter `points` given to `mgss()` before (controlling the accuracy of the computation).

`is.mixedDistribution`**:** A flag indicating whether the distribution is a mix of other loss distributions. The flag is set to `FALSE` by `lossDistribution()` and set to `TRUE` by `mgss()` and `mosg.lossDistribution()`.

`bw`**:** The bandwidth value that has been used to construct the loss distribution. For mixture distributions as usually constructed by `mgss()`, this value is set to the maximal `bw` values of all distributions in the mix.

`observations`**:** This contains the data from which the loss distributions have been constructed (either raw data or values of the density or cumulative distribution functions, for discrete distributions). The function `mgss()` sets this value to `NULL`.

`range`**:** The range of the data underlying the game (either the number of categories or the data range for continuous data).

`is.discrete` A flag indicating whether the distribution is categorical or continuous.

`normalizationFactor`**:** This field is mostly for internal purposes, and may in almost all cases not be required explicitly. Its inclusion in the output is, however, useful in occasions where the preference relation underneath the game shall be modified into a pure lexicographical order (see the *Hacks* Section 7.1). Yet it can be occasionally useful to modify.

Graphical Representation

Often it is very helpful to have a graphical representation besides the textual description. Figure 11 shows the result of `plot(eq)`. This plot is visually divided into the following areas:

plot of the optimal defense (bar- or density (line) plot)	
worst-case adversarial behavior for 1st goal	assured loss distr. for 1st goal
worst-case adversarial behavior for 2nd goal	assured loss distr. for 2nd goal
⋮	⋮
worst-case adversarial behavior for d-th goal	assured loss distr. for d-th goal

The mixed distributions come as bar plots, the respective (assured) loss distributions show up as either bar charts (for games with a discrete payoff structure) or as density line plots (for games with continuous distributions, like in Figure 7).

6. Customizing the Plots

The default behavior of all plotting routines is mostly unmodified from how R plots line or point charts. For game matrices, which are essentially matrices of plots, some manual fine-tuning of margins

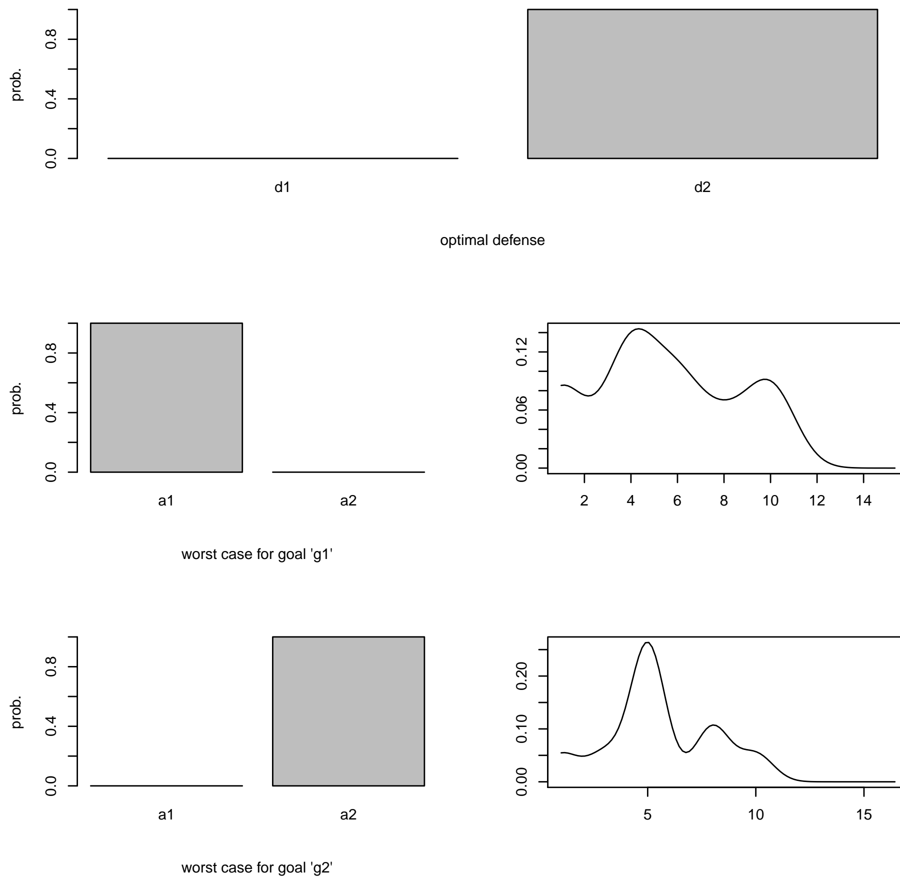


Figure 11: Graphical representation of an equilibrium

and axes ranges may be necessary. That is possible by setting the plot parameters externally using the `par` function (as shown in an example below), and by specifying the `xlim` and `ylim` parameters for `plot()`, with the same syntax and semantic as for any other plotting routine. We illustrate both in Section 6.2. These parameters are passed through to the underlying plot functions of R.

6.1. Plotting Loss Distributions

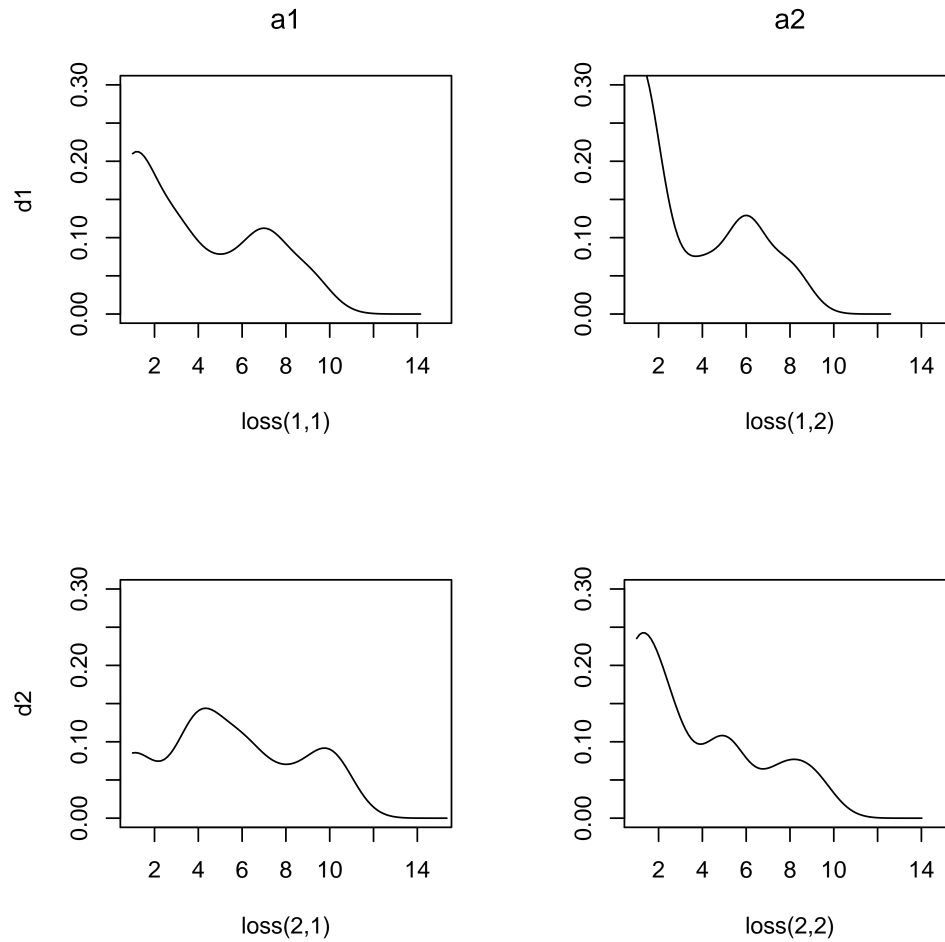
Plots of loss distributions allow customization by passing additional parameters to R's internal plot routines. Other annotations like legends or additional line plots work as with normal plots.

6.2. Plotting Game Matrices

Invoking the `plot()` function on a game produces a plot of the payoff matrix. Rows and columns of the matrix are labeled with the corresponding names of the defence and attack strategies. Each loss distribution is labeled with the indices (i, j) in the payoff matrix as it represents the loss in case of attack j when playing defense strategy i.

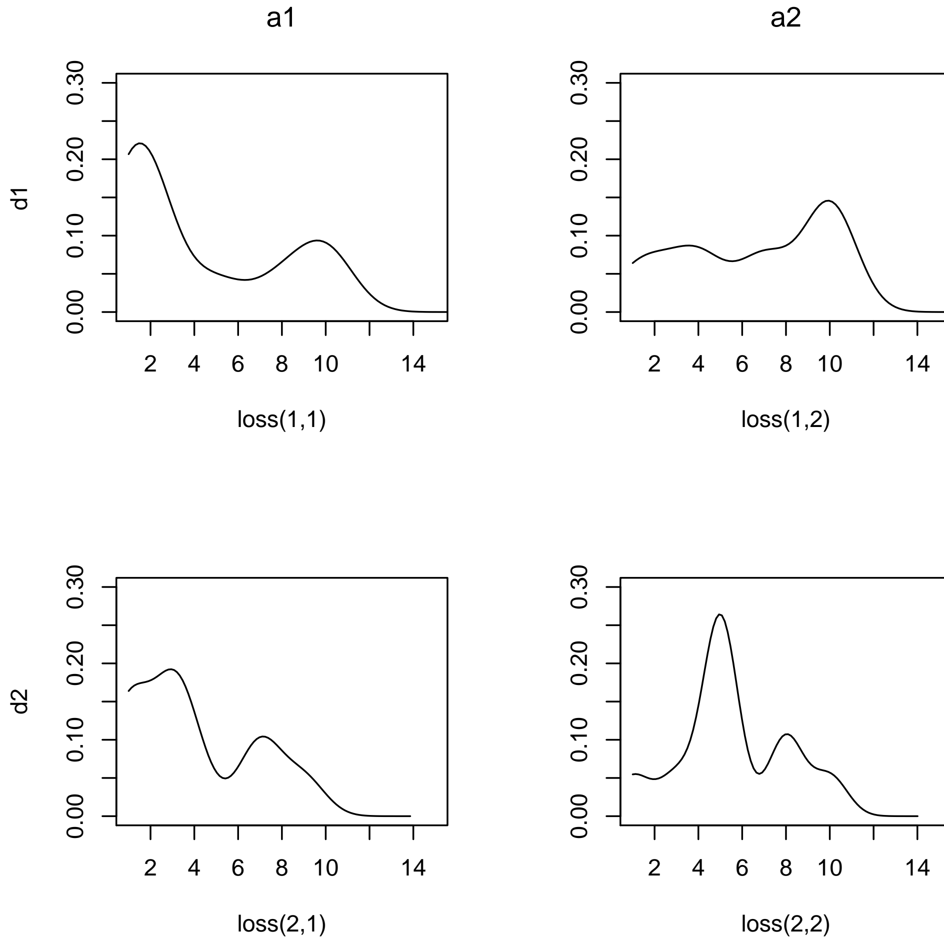
However, some care is needed in the case of several goals. Simply invoking `plot(G)` only returns a plot of the payoff matrix corresponding to the first goal, e.g., for the game from Figure 7

```
par(omi=c(0.5,0.5,0.5,0.5)) # adjust the margins
# use ylim and xlim to equalize the plots scales
plot(G1,xlim=c(1,15),ylim=c(0,0.3))
```



In order to see the payoffs related to other goals, the corresponding parameter `goal` needs to be used, e.g.,

```
plot(G1,goal=2,xlim=c(1,15),ylim=c(0,0.3))
```



to see the payoffs related to the second goal.

7. Remarks

7.1. Hacks

This section describes a few tricks to use the package in different ways or to overcome some of the (yet existing) limitations of it. However, all the methods described below must be used with the greatest care.

Using a Maximizing First Player

By default (and by the construction of the stochastic $\preceq$ ordering), player 1 is always minimizing.

For games defined over the reals (see Section 4.1), we can do a sign change to turn player 1 from minimizing into maximizing.

Less obvious is the case for a game over distributions. The key insight is that a player maximizing the distribution of a random variable X can equivalently minimize the distribution of the random variable $\max(\text{supp}(X)) - X + 1$. Observe that this is a simple *reversal* of the loss scale, with the "+1" only required to shift everything into the admissible range $[1, \infty)$. So, if we have raw data $= (x_1, x_2, \ldots)$ from which we construct empirical distributions, we just need to supply the data $\max(\boldsymbol{x}) - \boldsymbol{x} + 1$ instead when constructing the loss distributions. The game optimization will still go for the low losses, which, by the reversal, have now become relocated towards the right end of the scale.

For example, the game with the payoff matrix

$$A = \begin{pmatrix} 1 & 5 & 2 \\ 2 & 3 & -4 \\ 3 & 0 & 3 \end{pmatrix} \tag{7}$$

with a maximizing player has the Nash equilibrium $\boldsymbol{x}^* = (9/23, 3/46, 25/46) \approx (0.391, 0.065, 0.543)$, $\boldsymbol{y}^* = (16/23, 13/46, 1/46) \approx (0.695, 0.283, 0.022)$, computed using Gambit (McKelvey, McLennan, and Turocy 2007).

There are two ways of constructing this game in the package for a maximizing first player, both delivering the same results, which again are approximately equal to the exact equilibrium just given before.

```
# obvious method: use -A for the payoffs
version1 <- mosg(n = 3, m = 3, goals = 1,
    losses = list(-1, -5, -2, -2, -3, 4, -3, 0, -3))
mgss(version1)

##
##  equilibrium for multiobjective security game (MOSG)
##
## optimal defense strategy:
##         prob.
## 1 0.39130435
## 2 0.06521739
## 3 0.54347826
##
## worst case attack strategies per goal:
##             1
## 1 0.69565217
## 2 0.28260870
## 3 0.02173913

# general method: use max(A) - A for the payoffs
version2 <- mosg(n = 3, m = 3, goals = 1,
    losses = list(4, 0, 3, 3, 2, 9, 2, 5, 2))
mgss(version2)
```

```
##
##  equilibrium for multiobjective security game (MOSG)
##
## optimal defense strategy:
##        prob.
## 1 0.39130435
## 2 0.06521739
## 3 0.54347826
##
## worst case attack strategies per goal:
##            1
## 1 0.69565217
## 2 0.28260870
## 3 0.02173913
```

The roundoff errors already visible in this example will be revisited in Section 7.2, where we discuss their impact on the validity of the output as such.

Replacing the Stochastic by a Lexicographic Order

It is possible (both in theory and in practice) to play games not over stochastic orders, but uses a lexicographic order on the loss vectors. This may, for example, be relevant when a multi-objective optimization is such that the goals have a clear priority among them, where a player is first going for an optimal payoff in the first goal, and the second goal comes into play only to break a tie (likewise with the third, fourth and other goals). For risk management, it is interesting to fist optimize the *expected* damage, and only upon equality, go for the scenario with the smaller second moment (i.e., the variance). Next, if the variance is equal, we prefer the distribution that "leans" more towards lower damages, i.e., has smaller third moment (that relates to risk attitudes; see Wenner (2002)), and so on.

Further, it is possible to prepare the input for the games in a way that equalizes the stochastic $\preceq$ ordering with a lexicographic ordering on arbitrary values. Let us w.l.o.g. assume that two vectors $\boldsymbol{a} = (a_1, \dots, a_n)$ and $\boldsymbol{b} = (b_1, \dots, b_n)$ are given, and that we are interested in playing the game using the lexicographic order that first looks at (a_n, b_n), and upon equality continues to look at (a_{n-1}, b_{n-1}) and so on. For the example above, we may think of a_n, b_n as being means, a_{n-1}, b_{n-1} being variances, etc. Note that this way of comparing random variables is exactly the stochastic $\preceq$-ordering, by Lemma 1, as the densities are compared "pointwise" and in decreasing categorical ranks along the support.

The trick is a conversion of the vectors $\boldsymbol{a}, \boldsymbol{b}$ into new vectors $\boldsymbol{a}', \boldsymbol{b}'$ so that

$$\boldsymbol{a} <_{lex} \boldsymbol{b} \iff \boldsymbol{a}' \preceq \boldsymbol{b}' \tag{8}$$

in the stochastic order that the package uses. To this end, pick any sufficiently small value $\lambda > 0$ so that $\|\lambda \boldsymbol{a}\|_\infty < 1, \|\lambda \boldsymbol{b}\|_\infty < 1$, and $\|\lambda \boldsymbol{a}\|_1 < 1, \|\lambda \boldsymbol{b}\|_1 < 1$ and define

$$\boldsymbol{a}' := (1 - \|\lambda \boldsymbol{a}\|_1, \lambda \boldsymbol{a}), \quad \text{and } \boldsymbol{b}' := (1 - \|\lambda \boldsymbol{b}\|_1, \lambda \boldsymbol{b}). \tag{9}$$

By definition, $\boldsymbol{a}', \boldsymbol{b}'$ are both normalized vectors and hence can be taken as categorical distributions.

We described the conversion here for only two vectors, leaving the general case for $n > 2$ vectors as a straightforward extension. The only important fact along these lines is that all vectors in question are

scaled by the *same* $\lambda > 0$, which is easy to choose if there are only finitely many vectors, implied by the assumptions that games are finite.

As of version 2.0.0 of the package, one can supply vectors in the list of payoffs. The function `mosg()` internally recognizes this and constructs the respective payoff distributions to resemble the lexicographic ordering just as described above.

We will show that (8) holds under this construction.

"$\Rightarrow$": Let i be the largest index so that $a_j = b_j$ for $j > i$, i.e., i is the "earliest" coordinate of difference encountered when we start from the right. If $i > 1$, then $\boldsymbol{a} \preceq \boldsymbol{b}$, because we have $\lambda a_i < \lambda b_i$ and $\lambda > 0$. Since $\preceq$ is equivalent to a lexicographic ordering and the first index was not even looked at, the claim follows from Lemma 1.

Otherwise, if $i = 1$, then $a_j = b_j$ for all $j > 1$ and hence $\boldsymbol{a} = \boldsymbol{b}$. But in that case, we must also have $1 - \|\lambda \boldsymbol{a}\|_1 = 1 - \|\lambda \boldsymbol{b}\|_1$, so the vectors are identical and hence $\boldsymbol{a} \preceq \boldsymbol{b}$ holds with equality.

"$\Leftarrow$": Let i be defined as before. If $i > 1$, then $a'_i = \lambda a_i < b'_i = \lambda b_i$, using Lemma 1 again to describe $\preceq$ by a lexicographic order. In that case, however, we have an index in $\boldsymbol{a}, \boldsymbol{b}$ for which $a_i < b_i$, thus putting them in order $\boldsymbol{a} <_{lex} \boldsymbol{b}$, since $\lambda a_i = a'_j = b'_j = \lambda b_i$ for all $j > i$ and after dividing by $\lambda > 0$.

Otherwise, if $i = 1$, then $a'_i = b'_i$ by the same token as before, and we have $\boldsymbol{a}'$ identical to $\boldsymbol{b}'$ under the equivalence relation induced by $\preceq$. This implies $\boldsymbol{a} = \boldsymbol{b}$, so that the two also match lexicographically.

□

If a general game shall work with the lexicographic order of payoff vectors instead of distribution-valued payoffs, the procedure to use the package is thus the following:

1. Shift all payoffs into a strictly positive range, and find a *common* factor $\lambda > 0$ that scales all payoffs into the *half-open* unit interval $[0, 1)$.

2. Add an artificial "leftmost" category for the purpose of normalizing the resulting scaled vectors, i.e., write all payoff vectors $\boldsymbol{a}$ in the form $\boldsymbol{a}'$ as in (9).

3. For each such vector $\boldsymbol{a}'$, construct the payoff distribution by supplying $\boldsymbol{a}'$ as data to describe the probability density function, i.e., use

   ```
   lossDistribution(dat = a', dataType = "pdf",
                        smoothing = "none", ...)
   ```

4. Construct and solve the game as any other game, but bear in mind that the outputs of other functions like `mean()`, `quantile()` are all invalidated (as this is still a hack of the package). For the same reason, plots must be interpreted with care.

Using Degenerate Distributions with Non-Degenerates

Including degenerate distributions in the payoff structure *may* be meaningful if the goals that different kinds of distributions refer to are separated. That is, we may have one goal with all continuous distributions, coexisting with another goal that has all real-valued (degenerate) distributions, for example, if we want to optimize over damage distributions and also seeks to minimize disappointment rates, which is a real-valued payoff. The disappointment rate discussed by Wachter, Rass, König, and Schauer (2018) is here understood as the value $d = \Pr(X > \mathsf{E}(X))$, i.e., the chances that the game delivers more loss than expected (hence, "disappointing" player 1 in a round, though it may come out

opposite in other rounds; but this anyway meets player 1's expectation as an upper bound to the loss). In version 1.0.0.0 of the package, this is not supported, but the package can be tricked to accept such inputs.

Suppose that we have a real-valued payoff structure that shall be used with another that consists of distributions; depending on the type, the construction of the inputs proceeds differently. Without loss of generality, let us assume that the real-valued payoff structure has been scaled and shifted to be a matrix $\boldsymbol{A} \in [0,1)^{n\times m}$.

For discrete distributions over k categories, one can "embed" a real value a_{ij} into a discrete density function with masses $(\varepsilon, \varepsilon, \ldots, \varepsilon, 1-((k-2)\varepsilon + a_{ij}), a_{ij})$ for some sufficiently small ε that we choose to be the *same for all* a_{ij}. Like before, the stochastic order on the so-constructed probability mass functions (vectors) will then resemble the ordering of the real-valued payoffs in $\boldsymbol{A}$. In a way, this is a generalization of the Bernoulli representation that was used in Section 4.1, only extended to more than two categories, so that the representatives are compatible with the other payoff structures.

This trick fails for continuous distributions. In that case, we recommend adopting a "converse" approach: Theorem 14 in (Rass, König, and Schauer 2017) shows how to convert a distribution-valued game into a real-valued game whose equilibria are approximately equal, meaning that the equilibrium strategies in both games differ by no more than $\varepsilon > 0$ in the 1-norm; ε can herein be made as small as desired. Such a conversion then allows combining real-valued with distribution-valued payoff structures, at the price of resorting to another approximate version of the original game. This conversion, however, is currently not supported by the package.

7.2. Known Limitations and Issues

In version 1.0.0.0 of the package, known issues can broadly be divided into yet to be implemented functionality for the user, and numerical inaccuracy by roundoff errors. We shall discuss the latter in more detail below, and start with some convenience functions that are planned for inclusion in future versions.

Manipulating the Game Payoff Structure

Currently, games can only be manipulated by replacing single entries in the payoff matrix "manually". Say, to replace the ij-th entry for the k-th goal in the game `G`, we can compute the index

```
idx <- G$loc(i,j,k)
```

in the `losses` list, and then replace the entry accordingly by reassigning a new value:

```
G$losses[[idx]] <- new_loss_distribution
```

Bulk replacement or extraction syntax as for matrices (e.g., getting a whole row of `A` as `A[i,]` or similar) are not available for game structures at this point.

Plotting

Plotting large games may yield to undesirably small scaled plots in the matrices, so plotting games beyond five dimensions may not be too "visually expressive". However, it is possible to play around with the `par()` function to override the inner and outer margins; the `plot()` routines take these values as set externally.

8. Outlook

Future features planned include improvements on the known issues and limitations as outlined in Section 7.2, but also adding the possibility to specify a continuous distribution pointwise (so as to use a "named" loss distribution in the continuous setting).

References

Chu CY, Henderson D, Parmeter C (2015). "Plug-in Bandwidth Selection for Kernel Density Estimation with Discrete Data." *Econometrics*, **3**(2), 199–214. doi:10.3390/econometrics3020199. URL https://doi.org/10.3390/econometrics3020199.

Gouglidis A, König S, Green B, Rossegger K, Hutchison D (2018). "Protecting Water Utility Networks from Advanced Persistent Threats: A Case Study." In *Game Theory for Security and Risk Management*, pp. 313–333. Springer International Publishing. doi:10.1007/978-3-319-75268-6_13. URL https://doi.org/10.1007/978-3-319-75268-6_13.

Hayfield T, Racine JS (2008). "Nonparametric Econometrics: The np Package." *Journal of Statistical Software*, **27**(5). doi:10.18637/jss.v027.i05. URL https://doi.org/10.18637/jss.v027.i05.

International Standards Organisation (ISO) (2018). "ISO/IEC 31000 - Risk management – Principles and guidelines." Https://www.iso.org/standard/65694.html (accessed 27.06.2018).

Kiessé TS, Cuny HE (2013). "Discrete triangular associated kernel and bandwidth choices in semiparametric estimation for count data." *Journal of Statistical Computation and Simulation*, **84**(8), 1813–1829. doi:10.1080/00949655.2013.768995. URL https://doi.org/10.1080/00949655.2013.7689.

Klugman SA, Panjer HH, Willmot GE (1998). *Loss models: From data to decisions*. A Wiley-Interscience publication. Wiley, New York, NY. ISBN 0471238848. Panjer, Harry H. (VerfasserIn) and Willmot, Gordon E. (VerfasserIn), URL http://www.loc.gov/catdir/description/wiley031/97028718.html.

Kokonendji C, Kiessé TS, Zocchi SS (2007). "Discrete triangular distributions and non-parametric estimation for probability mass function." *Journal of Nonparametric Statistics*, **19**(6-8), 241–254. doi:10.1080/10485250701733.

König S, Gouglidis A, Green B, Solar A (2018). "Assessing the Impact of Malware Attacks in Utility Networks." In *Game Theory for Security and Risk Management*, pp. 335–351. Springer International Publishing. doi:10.1007/978-3-319-75268-6_14. URL https://doi.org/10.1007/978-3-319-75268-6_14.

Li Q, Racine J (2003). "Nonparametric estimation of distributions with categorical and continuous data." *Journal of Multivariate Analysis*, **86**(2), 266–292. doi:10.1016/s0047-259x(02)00025-8. URL https://doi.org/10.1016/s0047-259x(02)00025-8.

Lozovanu D, Solomon D, Zelikovsky A (2005). "Multiobjective Games and Determining Pareto-Nash Equilibria." *Buletinul Academiei de Stiinte a Republicii Moldova Matematica*, **3**(49), 115–122.

Matsushima H (1997). "Bounded Rationality in Economics: A Game Theorist's View." *The Japanese Economic Review*, **48**(3), 293–306. doi:10.1111/1468-5876.00056. URL https://doi.org/10.1111/1468-5876.00056.

McKelvey RD, McLennan AM, Turocy TL (2007). "Gambit: Software Tools for Game Theory, Version 0.2007.12.04." URL: http://gambit.sourceforge.net.

Nadaraya EA (1965). "On Non-Parametric Estimates of Density Functions and Regression Curves." *Theory of Probability & Its Applications*, **10**(1), 186–190. doi:10.1137/1110024.

Nash JF (1950). "Equilibrium points in n-person games." *Proceedings of the National Academy of Sciences*, **36**(1), 48–49. doi:10.1073/pnas.36.1.48. URL https://doi.org/10.1073/pnas.36.1.48.

R Development Core Team (2016). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

Rajagopalan B, Lall U (1995). "A kernel estimator for discrete distributions." *Journal of Nonparametric Statistics*, **4**(4), 409–426. doi:10.1080/10485259508832629.

Rass S (2013). "On Game-Theoretic Network Security Provisioning." *Journal of Network and Systems Management*, **21**(1), 47–64. doi:10.1007/s10922-012-9229-1. URL https://doi.org/10.1007/s10922-012-9229-1.

Rass S (2015). "On Game-Theoretic Risk Management (Part Two) – Algorithms to Compute Nash-Equilibria in Games with Distributions as Payoffs." ArXiv:1511.08591.

Rass S (2018). "Decision Making When Consequences Are Random." In *Game Theory for Security and Risk Management*, pp. 21–46. Springer International Publishing. doi:10.1007/978-3-319-75268-6_2. URL https://doi.org/10.1007/978-3-319-75268-6_2.

Rass S, König S (2019). "Package 'HyRiM': Multicriteria Risk Management using Zero-Sum Games with vector-valued payoffs that are probability distributions." https://cran.r-project.org/package=HyRiM. Version 1.0 (current stable release as of Jan.19; ongoing development).

Rass S, König S, Schauer S (2015). "Uncertainty in Games: Using Probability-Distributions as Payoffs." In *Lecture Notes in Computer Science*, pp. 346–357. Springer International Publishing. doi:10.1007/978-3-319-25594-1_20. URL https://doi.org/10.1007/978-3-319-25594-1_20.

Rass S, König S, Schauer S (2016). "Decisions with Uncertain Consequences—A Total Ordering on Loss-Distributions." *PLOS ONE*, **11**(12), e0168583. doi:10.1371/journal.pone.0168583. URL https://doi.org/10.1371/journal.pone.0168583.

Rass S, König S, Schauer S (2017). "Defending Against Advanced Persistent Threats Using Game-Theory." *PLoS ONE*, **12**(1), e0168675. doi:10.1371/journal.pone.0168675. Journal Article.

Rass S, Schauer S (eds.) (2018). *Game Theory for Security and Risk Management*. Springer International Publishing. doi:10.1007/978-3-319-75268-6. URL https://doi.org/10.1007/978-3-319-75268-6.

Schauer S, König S, Latzenhofer M, Rass S (2017). "Identifying and Managing Risks in Interconnected Utility Networks. The HyRiM Risk Management Process." In *The Eleventh International Conference on Emerging Security Information, Systems and Technologies (SECURWARE) 2017*, pp. 79–86. IARIA XPS Press. ISBN 978-1-61208-582-1.

Sela A (1999). "Fictitious play in 'one-against-all' multi-player games." *Economic Theory*, **14**(3), 635–651. doi:10.1007/s001990050345. URL https://doi.org/10.1007/s001990050345.

Selten R (1975). "A Re-Examination of the Perfectness Concept for Equilibrium Points in Extensive Games." *Int J Game Theory*, **4**(25), 25–55. doi:10.1007/BF01766400.

Shaked M, Shanthikumar JG (2006). *Stochastic Orders*. Springer.

Wachter J, Rass S, König S, Schauer S (2018). "Disappointment-Aversion in Security Games." In *Lecture Notes in Computer Science*, pp. 314–325. Springer International Publishing. doi:10.1007/978-3-030-01554-1_18. URL https://doi.org/10.1007/978-3-030-01554-1_18.

Wansouwé WE, Kokonendji CC, Kolyang D (2015). "Nonparametric estimation for probability mass function with Disake: an R package for discrete associated kernel estimators." *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées*, **19**, 1–23. URL https://hal.inria.fr/hal-01300100.

Wansouwé WE, Somé SM, Kokonendji CC (2016). "Ake: An R Package for Discrete and Continuous Associated Kernel Estimations." *The R Journal*, **8**(2), 258–276. URL https://journal.r-project.org/archive/2016/RJ-2016-045/index.html.

Wenner F (2002). *Determination of Risk Aversion and Moment-Preferences: A Comparison of Econometric models*. PhD Thesis, Universität St.Gallen.

Zougab N, Adjabi S, Kokonendji CC (2012). "Binomial kernel and Bayes local bandwidth in discrete function estimation." *Journal of Nonparametric Statistics*, **24**(3), 783–795. doi:10.1080/10485252.2012.678.