Package ‘GPGame’

March 22, 2018

Type Package
Title Solving Complex Game Problems using Gaussian Processes
Version 1.1.0
Date 2018-03-21
Author Victor Picheny, Mickael Binois
Maintainer Victor Picheny <victor.picheny@inra.fr>
Description
Sequential strategies for finding a game equilibrium are proposed in a black-box setting (expensive pay-off evaluations, no derivatives). The algorithm handles noiseless or noisy evaluations. Two acquisition functions are available. Graphical outputs can be generated automatically.
License GPL-3
LazyData TRUE
Imports Rcpp (>= 0.12.5), DiceKriging, GPareto, KrigInv, DiceDesign,
MASS, mnormt, mvtnorm, methods
Suggests DiceOptim
LinkingTo Rcpp
RoxygenNote 6.0.1
NeedsCompilation yes
Repository CRAN
Date/Publication 2018-03-22 10:52:19 UTC

R topics documented:
crit_PNash .......................................................... 2
crit_SUR_Eq ......................................................... 4
filter_for_Game ..................................................... 6
generate_integ_pts .............................................. 7
getEquilibrium ..................................................... 8
GPGame .......................................................... 11
nonDom .......................................................... 12
plotGame ........................................................ 14
crit_PNash

plotGameGrid .................................................. 15
solve_game .................................................. 17

Index 22

---

crit_PNash  

*Probability for a strategy of being a Nash Equilibrium*

---

**Description**

Acquisition function for solving game problems based on the probability for a strategy of being a Nash Equilibrium. The probability can be computed exactly using the multivariate Gaussian CDF (mnormt, pmvnorm) or by Monte Carlo.

**Usage**

```r
crit_PNash(idx, integcontrol, type = "simu", model, ncores = 1, control = list(nsim = 100, eps = 1e-06))
```

**Arguments**

- `idx` is the index on the grid of the strategy evaluated
- `integcontrol` is a list containing: `integ.pts`, a `[npts x dim]` matrix defining the grid, `expanded.indices` a matrix containing the indices of the `integ.pts` on the grid and `n.s`, a `nobj` vector containing the number of strategies per player
- `type` 'exact' or 'simu'
- `model` is a list of `nobj` km models
- `ncores` `mclapply` is used if > 1 for parallel evaluation
- `control` list with slots `nsim` (number of conditional simulations for computation) and `eps`
- `eps` numerical jitter for stability

**References**


**See Also**

`crit_SUR_Eq` for an alternative infill criterion
Examples

```r
# Not run:

## Example 1: 2 variables, 2 players, no filter

library(DiceKriging)
set.seed(42)

# Define objective function (R^2 -> R^2)
fun <- function(x)
{
  if (is.null(dim(x)))  x <- matrix(x, nrow = 1)
  b1 <- 15 * x[, 1] - 5
  b2 <- 15 * x[, 2]
  return(cbind((b2 - 5.1*(b1/(2*pi)))^2 + 5/pi*b1 - 6)^2 + 10*((1 - 1/(8*pi)) * cos(b1) + 1),
         -sqrt(((10.5 - b1)*(b1 + 5.5)*(b2 + 0.5)) - 1/30*(b2 - 5.1*(b1/(2*pi)))^2 - 6)^2 -
         1/3 * (((1 - 1/(8 * pi)) * cos(b1) + 1)))
}

# Grid definition
n.s <- rep(11, 2)
x.to.obj <- c(1,2)
gridtype <- 'cartesian'
integcontrol <- generate_integ_pts(n.s=n.s, d=2, nobj=2, x.to.obj = x.to.obj, gridtype=gridtype)
test.grid <- integcontrol$integ.pts
expanded.indices <- integcontrol$expanded.indices
n.init <- 11
design <- test.grid[sample.int(n=nrow(test.grid), size=n.init, replace=FALSE),]
response <- t(apply(design, 1, fun))
mf1 <- km(~., design = design, response = response[,1], lower=c(.1,.1))
mf2 <- km(~., design = design, response = response[,2], lower=c(.1,.1))
model <- list(mf1, mf2)

crit_sim <- crit_PNash(idx=1:nrow(test.grid), integcontrol=integcontrol,
                      type = "simu", model=model, control = list(nsim = 100))
crit_ex <- crit_PNash(idx=1:nrow(test.grid), integcontrol=integcontrol,
                      type = "exact", model=model)

filled.contour(seq(0, 1, length.out = n.s[1]), seq(0, 1, length.out = n.s[2]),
               zlim = c(0, 0.7),
               matrix(pmax(0, crit_sim), n.s[1], n.s[2]),
               main = "Pnash criterion (MC)",
               xlab = expression(x[1]), ylab = expression(x[2]),
               color = terrain.colors,
               plot.axes = (axis(1); axis(2));
               points(design[,1], design[,2], pch = 21, bg = "white")
)

filled.contour(seq(0, 1, length.out = n.s[1]), seq(0, 1, length.out = n.s[2]),
               zlim = c(0, 0.7),
               matrix(pmax(0, crit_ex), n.s[1], n.s[2]),
               main = "Pnash criterion (exact)",
               xlab = expression(x[1]), ylab = expression(x[2]),
               color = terrain.colors,
               plot.axes = (axis(1); axis(2));
               points(design[,1], design[,2], pch = 21, bg = "white")
)```
crit_SUR_Eq

Description

Computes the SUR criterion associated to an equilibrium for a given xnew and a set of trajectories of objective functions on a predefined grid.

Usage

crit_SUR_Eq(idx, model, integcontrol, Simu, precalc.data = NULL, equilibrium, n.ynew = NULL, cross = FALSE, IS = FALSE, plot = FALSE, kweights = NULL, Nadir = NULL)

Arguments

idx is the index on the grid of the strategy evaluated
model is a list of nobj km models
integcontrol is a list containing: integ.pts, a [npts x dim] matrix defining the grid, expanded.indices a matrix containing the indices of the integ.pts on the grid and n.s, a nobj vector containing the number of strategies per player
Simu is a matrix of size [npts x nsim*nobj] containing the trajectories of the objective functions (one column per trajectory, first all the trajectories for obj1, then obj2, etc.)
precalc.data is a list of length nobj of precalculated data (based on kriging models at integration points) for faster computation - computed if not provided
equilibrium equilibrium type: either "NE", "KSE", "CKSE" or "NKSE"
n.ynew is the number of ynew simulations (if not provided, equal to the number of trajectories)
cross if TRUE, all the combinations of trajectories are used (increases accuracy but also cost)
IS if TRUE, importance sampling is used for ynew
plot if TRUE, draws equilibria samples (should always be turned off)
kweights kriging weights for CKS (TESTING)
Nadir optional vector of size nobj. Replaces the nadir point for KSE. If only a subset of values needs to be defined, the other coordinates can be set to Inf.

References

crit_SUR_Eq

See Also
crit_PNash for an alternative infill criterion

Examples

```r
## Not run:
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# See Also
# Draw contour of the criterion
filled.contour(seq(0, 1, length.out = n.s[1]), seq(0, 1, length.out = n.s[2]),
    matrix(pmax(0, crit_grid), n.s[1], n.s[2]), main = "SUR criterion",
    xlab = expression(x[1]), ylab = expression(x[2]), color = terrain.colors,
    plot.axes = (axis(1)); axis(2);
    points design[,1], design[,2], pch = 21, bg = "white")
)

## End(Not run)

---

**filter_for_Game**  
*All-purpose filter*

### Description
Select candidate points for conditional simulations or for criterion evaluation, based on a "window" or a probability related to the equilibrium at hand.

### Usage
```r
filter_for_Game(n.s.target, model = NULL, predictions = NULL,
    type = "window", equilibrium = "NE", integcontrol, options = NULL,
    ncores = 1, random = TRUE, include.obs = FALSE, min.crit = 1e-12,
    nsamp = NULL, Nadir = NULL)
```

### Arguments
- `n.s.target` scalar or vector of number of strategies (one value per player) to select. For NE, if `n.s.target` is a scalar then each player will have \(\text{round}(n.s.target^{1/nobj})\) strategies.
- `model` is a list of `nobj` `nobj km` objects
- `predictions` is a list of size `nobj`
- `type` either "window", "PND" or "Pnash", see details
- `equilibrium` either 'NE', 'KSE' or 'NKSE' for Nash/Kalai-Smoridinsky/Nash-Kalai-Smoridinsky equilibria
- `integcontrol` is a list containing: `integ.pts`, a \([\text{npts} \times \text{dim}]\) matrix defining the grid, `expanded.indices` a matrix containing the indices of the `integ.pts` on the grid and `n.s`, a `nobj` vector containing the number of strategies per player
- `options` a list containing either the window (matrix or target) or the parameters for Pnash: method ("simu" or "exact") and `nsim`
- `ncores` `mclapply` is used if > 1 for parallel evaluation
**generate_integ_pts**

**random**
Boolean. If FALSE, the best points according to the filter criterion are chosen, otherwise the points are chosen by random sampling with weights proportional to the criterion.

**include.obs**
Boolean. If TRUE, the observations are included to the filtered set.

**min.crit**
Minimal value for the criterion, useful if random = TRUE.

**nsamp**
number of samples to estimate the probability of non-domination, useful when type='PND' and nobj>3.

**Nadir**
optional vector of size nobj. Replaces the nadir point for KSE. If only a subset of values needs to be defined, the other coordinates can be set to Inf.

**Details**
If type == "windows", points are ranked based on their distance to option$window (when it is a target vector), or based on the probability that the response belongs to option$window. The other options, "PND" (probability of non-domination, i.e., of not being dominated by the current Pareto front) and "Pnash" (probability of realizing a Nash equilibrium) base the ranking of points on the associated probability.

**Value**
List with two elements: I indices selected and crit the filter metric at all candidate points

---

**generate_integ_pts**  
*Strategy generation*

**Description**
Preprocessing to link strategies and designs.

**Usage**

```r
generate_integ_pts(n.s, d, nobj, x.to.obj = NULL, gridtype = "cartesian", 
equilibrium = "NE", lb = rep(0, d), ub = rep(1, d), 
include.obs = FALSE, model = NULL, seed = 42)
```

**Arguments**

- **n.s**
  scalar or vector. If scalar, total number of strategies (to be divided equally among players), otherwise number of strategies per player.

- **d**
  number of variables

- **nobj**
  number of objectives (or players)

- **x.to.obj**
  vector allocating variables to objectives. If not provided, default is 1:nobj, assuming that d=nobj

- **gridtype**
  either "cartesian" or "lhs", or a vector to define a different type for each player.
getEquilibrium

Equilibrium computation of a discrete game for a given matrix with objectives values

Description
Computes the equilibrium of three types of games, given a matrix of objectives (or a set of matrices) and the structure of the strategy space.

Usage
getEquilibrium(Z, equilibrium = c("NE", "NKSE", "KSE", "CKSE"), nobj = 2, n.s, expanded.indices = NULL, return.design = FALSE, sorted = FALSE, cross = FALSE, kweights = NULL, Nadir = NULL)
getEquilibrium

Arguments

- **Z** is a matrix of size \([npts \times \text{nsim} \times \text{nobj}]\) of objective values, see details.
- **equilibrium** considered type, one of "NE", "NKSE", "KSE", "CKSE"
- **nobj** nb of objectives (or players)
- **n.s** scalar of vector. If scalar, total number of strategies (to be divided equally among players), otherwise number of strategies per player.

expanded.indices

- is a matrix containing the indices of the integral points on the grid, see `generate_integ_pts`

return.design

- Boolean; if TRUE, the index of the optimal strategy is returned (otherwise only the pay-off is returned)

sorted

- Boolean; if TRUE, the last column of expanded.indices is assumed to be sorted in increasing order. This provides a substantial efficiency gain.

cross

- Should the simulation be crossed? (May be dropped in future versions)

kweights

- kriging weights for CKS (TESTING)

Nadir

- optional vector of size nobj. Replaces the nadir point for KSE. Some coordinates can be set to NULL.

Details

If nsim=1, each line of Z contains the pay-offs of the different players for a given strategy \(s: \text{[obj1(s), obj2(s), ...]}\). The position of the strategy \(s\) in the grid is given by the corresponding line of expanded.indices. If nsim>1, (vectorized call) Z contains different trajectories for each pay-off: each line is \([\text{obj1}_1(x), \text{obj1}_2(x), ..., \text{obj2}_1(x), \text{obj2}_2(x), ...]\).

Examples

```r
# Not run:
## Setup
fun <- function (x)
  {
    if (is.null(dim(x)))  x <- matrix(x, nrow = 1)
    b1 <- 15 * x[, 1] - 5
    b2 <- 15 * x[, 2]
    return(cbind(b2 - 5.1*(b1/(2*pi))^2 + 5/pi*b1 - 6)^2 + 10*((1 - 1/(8*pi)) * cos(b1) + 1),
           -sqrt((10.5 - b1)*(b1 + 5.5)*(b2 + 0.5) - 1/30*(b2 - 5.1*(b1/(2*pi))^2 - 6)^2 -
                  1/30 * ((1 - 1/(8 * pi)) * cos(b1) + 1)))
  }

d <- nobj <- 2

# Generate grid of strategies for Nash and Nash-Kalai-Smorodinsky
n.s <- c(11,11) # number of strategies per player
x.to.obj <- 1:2 # allocate objectives to players
integcontrol <- generate_integ_pts(n.s=n.s,d=d,nobj=nobj,x.to.obj=x.to.obj,gridtype="cartesian")
integ.pts <- integcontrol$integ.pts
expanded.indices <- integcontrol$expanded.indices

# Compute the pay-off on the grid
```
response.grid <- t(apply(integ.pts, 1, fun))

# Compute the Nash equilibrium (NE)
trueEq <- getEquilibrium(Z = response.grid, equilibrium = "NE", nobj = nobj, n.s = n.s,
  return.design = TRUE, expanded.indices = expanded.indices,
  sorted = !is.unsorted(expanded.indices[,2])))

# Pay-off at equilibrium
print(trueEq$NE)  

# Optimal strategy
print(integ.pts[trueEq$NE,])

# Index of the optimal strategy in the grid
print(expanded.indices[trueEq$NE,])

# Plots
par(mfrow = c(1,2))
plotGameGrid(fun = fun, n.grid = n.s, x.to.obj = x.to.obj, integcontrol=integcontrol,
  equilibrium = "NE")

# Compute KS equilibrium (KSE)
trueKSeq <- getEquilibrium(Z = response.grid, equilibrium = "KSE", nobj = nobj,
  return.design = TRUE, sorted = !is.unsorted(expanded.indices[,2])))

# Pay-off at equilibrium
print(trueKSeq$NE)  

# Optimal strategy
print(integ.pts[trueKSeq$NE,])

plotGameGrid(fun = fun, n.grid = n.s, integcontrol=integcontrol,
  equilibrium = "KSE", fun.grid = response.grid)

# Compute the Nash equilibrium (NE)
trueNKSeq <- getEquilibrium(Z = response.grid, equilibrium = "NKSE", nobj = nobj, n.s = n.s,
  return.design = TRUE, expanded.indices = expanded.indices,
  sorted = !is.unsorted(expanded.indices[,2])))

# Pay-off at equilibrium
print(trueNKSeq$NE)  

# Optimal strategy
print(integ.pts[trueNKSeq$NE,])

# Index of the optimal strategy in the grid
print(expanded.indices[trueNKSeq$NE,])

# Plots
plotGameGrid(fun = fun, n.grid = n.s, x.to.obj = x.to.obj, integcontrol=integcontrol,
  equilibrium = "NKSE")

## End(Not run)
Description

Sequential strategies for finding game equilibria in a black-box setting (expensive pay-off evaluations, no derivatives). Handles noiseless or noisy evaluations. Two acquisition functions are available. Graphical outputs can be generated automatically.

Details

Important functions:
solve_game
plot_game

Author(s)

Victor Picheny, Mickael Binois

References


See Also

DiceKriging, DiceOptim, KrigInv, GPareto

Examples

```R
# Not run:
# To use parallel computation (turn off on Windows)
library(parallel)
parallel <- FALSE # TRUE #
if(parallel) ncores <- detectCores() else ncores <- 1

#########################################################################
# 2 variables, 2 players, Nash equilibrium
# Player 1 (P1) wants to minimize fun1 and player 2 (P2) fun2
# P1 chooses x2 and P2 x2

#########################################################################
# First, define objective function fun: (x1,x2) -> (fun1,fun2)
fun <- function (x)
{
```
if (is.null(dim(x)))  x <- matrix(x, nrow = 1)
b1 <- 15 * x[, 1] - 5
b2 <- 15 * x[, 2]
return(cbind((b2 - 5.1*(b1/(2*pi)))^2 + 5/pi*b1 - 6)^2 + 10*(((1 - 1/(8*pi)) * cos(b1) + 1),
-sqrt(((10.5 - b1)*((b1 + 5.5)*(b2 + 0.5)) - 1/30*(b2 - 5.1*(b1/(2*pi)))^2 - 6)^2 -
1/3 * (((1 - 1/(8 * pi)) * cos(b1) + 1)))
}

# x.to.obj indicates that P1 chooses x1 and P2 chooses x2
x.to.obj <- c(1,2)

# Define a discretization of the problem: each player can choose between 21 strategies
# The ensemble of combined strategies is a 21x21 cartesian grid

# n.s is the number of strategies (vector)
n.s <- rep(21, 2)
# gridtype is the type of discretization
gridtype <- 'cartesian'
integcontrol <- list(n.s=n.s, gridtype=gridtype)

# Run solver with 6 initial points, 14 iterations
n.init <- 6 # number of initial points (space-filling)
n.ite <- 14 # number of iterations (sequential infill points)
res <- solve_game(fun, equilibrium = "NE", crit = "sur", n.init=n.init, n.ite=n.ite,
  d = 2, nobj=2, x.to.obj = x.to.obj, integcontrol=integcontrol,
  ncores = ncores, trace=1, seed=1)

# Get estimated equilibrium and corresponding pay-off
NE <- res$Eq.design
Poff <- res$Eq.poff

# Draw results
plotGame(res)

# See solve_game for other examples

## End(Not run)
nonDom

Description
Extract non-dominated points from a set, or with respect to a reference Pareto front

Usage
nonDom(points, ref = NULL, return.idx = FALSE)

Arguments
points matrix (one point per row) from which to extract non-dominated points, or, if a reference ref is provided, non-dominated points with respect to ref
ref matrix (one point per row) of reference (faster if they are already Pareto optimal)
return.idx if TRUE, return indices instead of points

Details
Use Kung non-domination sorting

Value
Non-dominated points from points, unless a ref is provided, in which case return points from points non-dominated by ref. If return.idx is TRUE, only returns indices

References

Examples
```r
## Not run:
d <- 6
n <- 1000
n2 <- 1000
test <- matrix(runif(d * n), n)
ref <- matrix(runif(d * n), n)
indPF <- nonDom(ref, return.idx = TRUE)
all(nonDom(ref) == ref[indPF,])

system.time(res <- nonDom(test, ref[indPF,, drop = FALSE], return.idx = TRUE))

res2 <- rep(NA, n2)
library(eomoa)
t0 <- Sys.time()
for(i in 1:n2){
  res2[i] <- !is_dominated(t(rbind(test[i, drop = FALSE], ref[indPF,])))[1]
}
print(Sys.time() - t0)
```
all(res == which(res2))

all(nonDom(test, ref) == test[res2,])

## End(Not run)

---

plotGame                     Plot equilibrium search result (2-objectives only)

### Description
Plot equilibrium search result (2-objectives only)

### Usage
```
plotGame(res, equilibrium = "NE", add = FALSE, UQ_eq = TRUE,
         simus = NULL, integcontrol = NULL, simucontrol = NULL, Nadir = NULL,
         ncores = 1)
```

### Arguments
- `res`: list returned by `solve_game`
- `equilibrium`: either "NE" for Nash, "KSE" for Kalai-Smoridinsky and "NKSE" for Nash-Kalai-Smoridinsky
- `add`: logical; if TRUE adds the first graphical output to an already existing plot; if FALSE, (default) starts a new plot
- `UQ_eq`: logical; should simulations of the equilibrium be displayed?
- `simus`: optional matrix of conditional simulation if `UQ_eq` is TRUE
- `integcontrol`: list with n.s element (maybe n.s should be returned by solve_game). See `solve_game`.  
- `simucontrol`: optional list for handling conditional simulations. See `solve_game`.  
- `Nadir`: optional vector of size nobj. Replaces the nadir point for KSE. If only a subset of values needs to be defined, the other coordinates can be set to Inf.
- `ncores`: number of CPU available (> 1 makes mean parallel TRUE)

### Examples
```
## Not run:
library(GPareto)
library(parallel)

# Turn off on Windows
parallel <- FALSE # TRUE
ncores <- 1
if(parallel) ncores <- detectCores()
```
plotGameGrid

Description

Plot equilibrium for 2 objectives test problems with evaluations on a grid. The number of variables is not limited.
Usage

plotGameGrid(fun = NULL, domain = NULL, n.grid, graphs = c("both", "design", "objective"), x.to.obj = NULL, integ.control = NULL, equilibrium = c("NE", "KSE", "CKSE", "NKSE"), fun.grid = NULL, Nadir = NULL, ...)

Arguments

fun
name of the function considered

domain
optional matrix for the bounds of the domain (for now \([0,1]^d\) only), (two columns matrix with min and max)

n.grid
number of divisions of the grid in each dimension (must correspond to n.s for Nash equilibriums)

graphs
either "design", "objective" or "both" (default) for which graph to display

x.to.obj, integ.control
see `solve_game` (for Nash equilibrium only)

equilibrium
either "NE" for Nash, "KSE" for Kalai-Smoridinsky and "NKSE" for Nash-Kalai-Smoridinsky

fun.grid
optional matrix containing the values of fun at integ.pts. Computed if not provided.

Nadir
optional vector of size nobj. Replaces the nadir point for KSE. If only a subset of values needs to be defined, the other coordinates can be set to Inf.

... further arguments to fun

Value

list returned by invisible() with elements:

- trueEqdesign design corresponding to equilibrium value trueEq
- trueEqPoff corresponding values of the objective
- trueParetoFront Pareto front
- response.grid
- integ.pts, expanded.indices

Examples

```r
# Not run:
library(GPareto)

# 2 variables
dom <- matrix(c(0,0,1,1),2)

plotGameGrid("P1", domain = dom, n.grid = 51, equilibrium = "NE")
plotGameGrid("P1", domain = dom, n.grid = rep(31,2), equilibrium = "NE")  # As in the tests
plotGameGrid("P1", domain = dom, n.grid = 51, equilibrium = "KSE")
plotGameGrid("P1", domain = dom, n.grid = rep(31,2), equilibrium = "NKSE")
```
solve_game

plotGameGrid("P1", graphs = "design", domain = dom, n.grid = rep(31,2), equilibrium = "NKSE")

## 4 variables

```r
dom <- matrix(rep(c(0,1), each = 4), 4)
plotGameGrid("ZDT3", domain = dom, n.grid = 25, equilibrium = "NE", x.to.obj = c(1,1,2,2))
```

## End(Not run)

---

solve_game

Main solver

Description

Main function to solve games.

Usage

```r
solve_game(fun, ..., equilibrium = "NE", crit = "sur", model = NULL,
  n.init = NULL, n.ite, d, nobj, x.to.obj = NULL, noise.var = NULL,
  Nadir = NULL, integcontrol = NULL, simucontrol = NULL,
  filtercontrol = NULL, kmcontrol = NULL, returncontrol = NULL,
  ncores = 1, trace = 1, seed = NULL)
```

Arguments

- **fun**: function with vectorial output
- **...**: additional parameter to be passed to fun
- **equilibrium**: either 'NE', 'KSE', 'CKSE' or 'NKSE' for Nash / Kalai-Smorodinsky / Copula-Kalai-Smorodinsky / Nash-Kalai-Smorodinsky equilibria
- **crit**: 'sur' (default) is available for all equilibria, 'psim' and 'pex' are available for Nash
- **model**: list of km models
- **n.init**: number of points of the initial design of experiments if no model is given
- **n.ite**: number of iterations of sequential optimization
- **d**: variable dimension
- **nobj**: number of objectives (players)
- **x.to.obj**: for NE and NKSE, which variables for which objective
- **noise.var**: noise variance. Either a scalar (same noise for all objectives), a vector (constant noise, different for each objective), a function (type closure) with vectorial output (variable noise, different for each objective) or "given_by_fn", see Details. If not provided, noise.var is taken as the average of model@noise.var.
- **Nadir**: optional vector of size nobj. Replaces the nadir point for KSE. If only a subset of values needs to be defined, the other coordinates can be set to Inf.


integcontrol  optional list for handling integration points. See Details.

simucontrol  optional list for handling conditional simulations. See Details.

filtercontrol  optional list for handling filters. See Details.

kmcontrol  optional list for handling km models. See Details.

returncontrol  optional list for choosing return options. See Details.

ncores  number of CPU available (> 1 makes mean parallel TRUE)

trace  controls the level of printing: 0 (no printing), 1 (minimal printing), 3 (detailed printing)

seed  to fix the random variable generator

Details

If noise.var="given_by_fn", fn returns a list of two vectors, the first being the objective functions and the second the corresponding noise variances.

integcontrol controls the way the design space is discretized. One can directly provide a set of points integ.pts with corresponding indices expanded.indices (for NE). Otherwise, the points are generated according to the number of strategies n.s. If n.s is a scalar, it corresponds to the total number of strategies (to be divided equally among players), otherwise it corresponds to the nb of strategies per player. In addition, one may choose the type of discretization with gridtype. Options are 'lhs' or 'cartesian'. Finally, lb and ub are vectors specifying the bounds for the design variables. By default the design space is [0,1]^d. A renew slot is available, if TRUE, then integ.pts are changed at each iteration. Available only for KSE and CKSE. For CKSE, setting the slot kweights=TRUE allows to increase the number of integration points, with nsamp (default to 1e4) virtual simulation points.

simucontrol controls options on conditional GP simulations. Options are IS: if TRUE, importance sampling is used for ynew; n.ynew number of samples of y(x_{n+1}) and n.sim number of sample path generated.

filtercontrol controls filtering options. filter sets how to select a subset of simulation and candidate points, either either a single value or a vector of two to use different filters for simulation and candidate points. Possible values are 'window', 'Pnash' (for NE), 'PND' (probability of non domination), 'none'. nsimPoints and ncandPoints set the maximum number of simulation/candidate points wanted (use with filter 'Pnash' for now). Default values are 800 and 200, resp. randomFilter (TRUE by default except for filter window) sets whereas the filter acts randomly or deterministically. For more than 3 objectives, PND is estimated by sampling; the number of samples is controled by nsamp (default to max(20, 5 * nobj)).

kmcontrol Options for handling nobj km models. cov.reestim (Boolean, TRUE by default) specifies if the kriging hyperparameters should be re-estimated at each iteration,

returncontrol sets options for the last iterations and what is returned by the algorithm. track.Eq allows to estimate the equilibrium at each iteration; options are 'none' to do nothing, 'mean' (default) to compute the equilibrium of the prediction mean (all candidates), "empirical" (for KSE) and "pex"/"psim" (NE only) for using Pnash estimate (along with mean estimate, on integ.pts only. NOT reestimated if filter.simu or crit is Pnash). The boolean force.exploit.last (default to TRUE) allows to evaluate the equilibrium on the predictive mean - if not already evaluated - instead of using crit (i.e., sur) for KSE and CKSE.
Value

A list with components:

- **model**: a list of objects of class `km` corresponding to the last kriging models fitted.
- **jplus**: recorded values of the acquisition function maximizer
- **integ.pts and expanded.Indices**: the discrete space used,
- **predEq**: a list containing the recorded values of the estimated best solution,
- **Eq.design, Eq.poff**: estimated equilibrium and corresponding pay-off

Note: with CKSE, kweights are not used when the mean on integ.pts is used

References


Examples

```r
## Not run:

# Example 1: Nash equilibrium, 2 variables, 2 players, no filter

# Define objective function (R^2 -> R^2)
fun1 <- function(x)
{
  if (is.null(dim(x))) x <- matrix(x, nrow = 1)
  b1 <- 15 * x[, 1] - 5
  b2 <- 15 * x[, 2]
  return(cbind((b2 - 5.1*(b1/(2*pi))^2 + 5/pi*b1 - 6)^2 + 10*((1 - 1/(8*pi)) * cos(b1) + 1),
              -sqrt(((10.5 - b1)*(b1 + 5.5)*(b2 + 0.5)) - 1/30*(b2 - 5.1*(b1/(2*pi))^2 - 6)^2 - 1/3 * ((1 - 1/(8 * pi)) * cos(b1) + 1))))
}

# To use parallel computation (turn off on Windows)
library(parallel)
parallel <- FALSE # TRUE
if(parallel) ncores <- detectCores() else ncores <- 1

# Grid definition
n.s <- rep(21, 2)
x.to.obj <- c(1,2)
gridtype <- 'cartesian'

# Run solver with 6 initial points, 4 iterations
# Increase n.ite to at least 10 for better results
res <- solve_game(fun1, equilibrium = "NE", crit = "sur", n.init=6, n.ite=4,
                   d = 2, nobj=2, x.to.obj = x.to.obj,
                   integcontrol=list(n.s=n.s, gridtype=gridtype),
                   ...)```
ncores = ncores, trace=1, seed=1)

# Get estimated equilibrium and corresponding pay-off
NE <- res$Eq.design
Poff <- resEq.poff

# Draw results
plotGame(res)

# Example 2: same example, KS equilibrium with given Nadir
# Run solver with 6 initial points, 4 iterations
# Increase n.ite to at least 10 for better results
res <- solve_game(fun1, equilibrium = "KSE", crit = "sur", n.init=6, n.ite=4,
                  d = 2, nobj=2, x.to.obj = x.to.obj,
                  integcontrol=list(n.s=400, gridtype="lhs"),
                  ncores = ncores, trace=1, seed=1, Nadir=c(Inf, -20))

# Get estimated equilibrium and corresponding pay-off
NE <- res$Eq.design
Poff <- resEq.poff

# Draw results
plotGame(res, equilibrium = "KSE", Nadir=c(Inf, -20))

# Example 3: Nash equilibrium, 4 variables, 2 players, filtering

fun2 <- function(x, nobj = 2){
  if (is.null(dim(x)))  x <- matrix(x, 1)
  y <- matrix(x[, 1:(nobj - 1)], nrow(x))
  z <- matrix(x[, nobj:ncol(x)], nrow(x))
  g <- rowSums((z - 0.5)^2)
  tmp <- t(apply(cos(y * pi/2), 1, cumprod))
  tmp <- cbind(t(apply(tmp, 1, rev)), 1)
  tmp2 <- cbind(1, t(apply(sin(y * pi/2), 1, rev)))
  return(tmp * tmp2 * (1 + g))
}

# Grid definition: player 1 plays x1 and x2, player 2 x3 and x4
# The grid is a lattice made of two LHS designs of different sizes
n.s <- c(44, 43)
x.to.obj <- c(1,1,2,2)
gridtype <- 'lhs'

# Set filtercontrol: window filter applied for integration and candidate points
# 500 simulation and 200 candidate points are retained.
filtercontrol <- list(msimPoints=500, ncandPoints=200, 
    filter=c("window", "window"))

# Set km control: lower bound is specified for the covariance range
# Covariance type and model trend are specified
solve_game

```
kmcontrol <- list(lb=rep(.2,4), model.trend=-1, covtype="matern3_2")

# Run solver with 20 initial points, 4 iterations
# Increase n.ite to at least 20 for better results
res <- solve_game(fun2, equilibrium = "NE", crit = "psim", n.init=20, n.ite=2,
   d = 4, nobj=2, x.to.obj = x.to.obj,
   integcontrol=list(n.s=n.s, gridtype=gridtype),
   filtercontrol=filtercontrol,
   kmcontrol=kmcontrol,
   ncores = 1, trace=1, seed=1)

# Get estimated equilibrium and corresponding pay-off
NE <- res$Eq.design
Poff <- res$Eq.poff

# Draw results
plotGame(res)

# Example 4: same example, KS equilibrium
# Grid definition: simple lhs
integcontrol=list(n.s=1e4, gridtype='lhs')

# Run solver with 20 initial points, 4 iterations
# Increase n.ite to at least 20 for better results
res <- solve_game(fun2, equilibrium = "KSE", crit = "sur", n.init=20, n.ite=2,
   d = 4, nobj=2,
   integcontrol=integcontrol,
   filtercontrol=filtercontrol,
   kmcontrol=kmcontrol,
   ncores = 1, trace=1, seed=1)

# Get estimated equilibrium and corresponding pay-off
NE <- res$Eq.design
Poff <- res$Eq.poff

# Draw results
plotGame(res, equilibrium = "KSE")

## End(Not run)
```
Index

crit_PNash, 2, 5
crit_SUR_Eq, 2, 4

DiceKriging, 11
DiceOptim, 11

filter_for_Game, 6

generate_integ_pts, 7, 9
getEquilibrium, 8
G Pareto, 11
GPGame, 11

km, 2, 4, 6, 17–19
KrigInv, 11

lhsDesign, 8

mclapply, 2, 6

nonDom, 12

plotGame, 11, 14
plotGameGrid, 15

solve_game, 11, 14, 16, 17