Package 'GPGame'

June 10, 2017

Type Package

Title Solving Complex Game Problems using Gaussian Processes

Version 1.0.0

Date 2017-06-09

Author Victor Picheny, Mickael Binois

Maintainer Victor Picheny <victor.picheny@inra.fr>

Description Sequential strategies for finding game equilibria 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, emoa, methods

LinkingTo Rcpp

RoxygenNote 5.0.1

NeedsCompilation yes

Repository CRAN

Date/Publication 2017-06-10 06:17:19 UTC

R topics documented:

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 mutivariate Gaussian CDF (mnormt, pmvnorm) or by Monte Carlo.

Usage

```
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 containting 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

V. Picheny, M. Binois, A. Habbal (2016+), A Bayesian optimization approach to find Nash equilibria, *https://arxiv.org/abs/1611.02440*.

See Also

crit_SUR_Eq for an alternative infill criterion

Examples

```
## 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")
                           }
)

## End(Not run)
```

---

crit_SUR_Eq *SUR criterion for equilibria*

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)
```

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" 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)

References

V. Picheny, M. Binois, A. Habbal (2016+), A Bayesian optimization approach to find Nash equilibria, *https://arxiv.org/abs/1611.02440*.

See Also

crit_PNash for an alternative infill criterion

Examples

```
## Not run:
##############################################
# 2 variables, 2 players
##############################################
library(DiceKriging)
set.seed(42)
```

```
# 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(14, 2)
x.to.obj   <- c(1,2)
gridtype <- 'cartesian'
integcontrol <- generate_integ_pts(n.s=n.s, d=4, nobj=2, x.to.obj = x.to.obj, gridtype=gridtype)
integ.pts <- integcontrol$integ.pts
expanded.indices <- integcontrol$expanded.indices

# Kriging models
n.init <- 11
design <- integ.pts[sample.int(n=nrow(integ.pts), 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)

# Conditional simulations
Simu <- t(Reduce(rbind, lapply(model, simulate, nsim=10, newdata=integ.pts, cond=TRUE,
                                   checkNames=FALSE, nugget.sim = 10^-8)))

# Useful precalculations
library(KrigInv)
precalc.data <- lapply(model, FUN=KrigInv:::precomputeUpdateData, integration.points=integ.pts)

# Compute criterion for all points on the grid
crit_grid <- lapply(X=1:prod(n.s), FUN=crit_SUR_Eq, model=model,
                    integcontrol=integcontrol, equilibrium = "NE",
                  Simu=Simu, precalc.data=precalc.data, n.ynew=10, IS=FALSE, cross=FALSE)
crit_grid <- unlist(crit_grid)

# 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

```
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)
```

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 `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 [`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 containting 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
`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`.

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

```
generate_integ_pts(n.s, d, nobj, x.to.obj = NULL, gridtype = "cartesian",
  equilibrium = "NE", lb = rep(0, d), ub = rep(1, d))
```

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.
`equilibrium`	either `"NE"`, `"KSE"` or `"NKSE"`
`lb, ub`	vectors specifying the bounds of the design space, by default `[0,1]^d`

Value

A list containing two matrices, `integ.pts` the design of experiments and `expanded.indices` the corresponding indices (for `NE`), and the vector `n.s`

Examples

```
## Not run:
##############################################
### 4 variables, 2 players, no filter
##############################################

# Create a 11x8 lattice based on 2 LHS designs
n.s <- c(11,8)
gridtype = "lhs"
# 4D space is split in 2
x.to.obj <- c(1,1,2,2)
integcontrol <- generate_integ_pts(n.s=n.s, d=4, nobj=2, x.to.obj = x.to.obj, gridtype=gridtype)
```

```
pairs(integcontrol$integ.pts)

## End(Not run)
```

`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"), nobj = 2, n.s,
  expanded.indices = NULL, return.design = FALSE, sorted = FALSE,
  cross = FALSE)
```

Arguments

`Z`	is a matrix of size [`npts x nsim*nobj`] of objective values, see details,
`equilibrium`	considered type, one of `"NE"`, `"NKSE"`, `"KSE"`
`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 `integ.pts` 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)

Details

If `nsim=1`, each line of `Z` contains the pay-offs of the different players for a given strategy s: [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 [obj1_1(x), obj1_2(x), ... obj2_1(x), obj2_2(x), ...].

Examples

```
## 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-
                1/3 * ((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$NEPoff)

# 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$NEPoff)

# 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$NEPoff)

# 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)
```

GPGame *Package GPGame*

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
plotGame

Author(s)

Victor Picheny, Mickael Binois

References

V. Picheny, M. Binois, A. Habbal (2016+), A Bayesian optimization approach to find Nash equilibria, *https://arxiv.org/abs/1611.02440*.

See Also

DiceKriging, DiceOptim, KrigInv, GPareto

Examples

```
## 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)
```

---

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

Examples

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

# Turn off on Windows
parallel <- TRUE # FALSE #
ncores <- 1
if(parallel) ncores <- detectCores()
cov.reestim <- TRUE
n.sim <- 20
n.ynew <- 20
IS <- TRUE
set.seed(1)

pb <- "P1" # 'P1' 'PDE' 'Diff'
fun <- P1

equilibrium = "NE"

d <- 2
nobj <- 2
n.init <- 20
n.ite <- 4
model.trend <- ~1
n.s <- rep(31, 2) #31
x.to.obj   <- c(1,2)
gridtype <- 'cartesian'
nsimPoints <- 800
ncandPoints <- 200
sur_window_filter <- NULL
sur_pnash_filter  <- NULL
Pnash_only_filter <- NULL
res <- solve_game(fun, equilibrium = equilibrium, crit = "sur", model = NULL, n.init=n.init,
  n.ite = n.ite, nobj=nobj, x.to.obj = x.to.obj, integcontrol=list(n.s=n.s, gridtype=gridtype),
  simucontrol=list(n.ynew=n.ynew, n.sim=n.sim, IS=IS), ncores = ncores, d = d,
  filtercontrol=list(filter=sur_window_filter, nsimPoints=nsimPoints, ncandPoints=ncandPoints),
  kmcontrol=list(model.trend=model.trend), trace=3,
  seed=1)
plotGame(res, equilibrium = equilibrium)

dom <- matrix(c(0,0,1,1),2)
plotGameGrid("P1", graphs = "objective", domain = dom, n.grid = 51, equilibrium = equilibrium)
plotGame(res, equilibrium = equilibrium, add = TRUE)



## End(Not run)
```

---

plotGameGrid *Visualisation of equilibrium solution in input/output space*

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, integcontrol = NULL,
  equilibrium = c("NE", "KSE", "NKSE"), fun.grid = 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, integcontrol	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.
...	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

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

## 4 variables
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

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

Arguments

`fun`	fonction with vectorial output
`...`	additional parameter to be passed to `fun`
`equilibrium`	either 'NE', 'KSE' or 'NKSE' for Nash/Kalai-Smoridinsky/Nash-Kalai-Smoridinsky 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`.
`integcontrol`	optional list for handling integration points. See Details.

simucontrol	optional list for handling conditional simulations. See Details.
plotcontrol	optional list for handling during-optimization plots. 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.

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.

plotcontrol can be used to generate plots during the search. Options are plots (Boolean, FALSE by default), compute.actual (Boolean, FALSE by default, to draw the actual problem, only for inexpensive fun), and pbname (string, for figure title and pdf export).

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) sets whereas the filter acts randomly or deterministically.

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. return.Eq (Boolean, TRUE by default) specifies if a final search for the equilibrium is performed at the end. finalcrit sets a different criterion for the last iteration. 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).

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 (if `return.Eq==TRUE`)

References

V. Picheny, M. Binois, A. Habbal (2016+), A Bayesian optimization approach to find Nash equilibria, *https://arxiv.org/abs/1611.02440*.

Examples

```
## Not run:

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

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

# Simple configuration: no filter, discretization is a 21x21 grid

# 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(fun, 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$Eq.design
Poff <- res$Eq.poff

# Draw results
plotGame(res)

##############################################
# Example 2: 4 variables, 2 players, filtering
##############################################
fun <- 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(nsimPoints=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
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(fun, 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)
```

```
## End(Not run)
```

Index