Package ‘DiceOptim’

October 12, 2022

Version 2.1.1
Title Kriging-Based Optimization for Computer Experiments
Date 2021-01-29
Description Efficient Global Optimization (EGO) algorithm as described in `Rous-
tant et al. (2012)` <doi:10.18637/jss.v051.i01> and adaptations for prob-
lems with noise (`Picheny and Ginsbourger, 2012`) <doi:10.1016/j.csda.2013.03.018>, paral-
lel infill, and problems with constraints.
Depends DiceKriging (>= 1.2), methods
Imports randtoolbox, pbivnorm, rgenoud, mnormt, DiceDesign, parallel
Suggests KrigInv, GPareto
License GPL-2 | GPL-3
URL http://dice.emse.fr/
RoxygenNote 7.1.1
NeedsCompilation yes
Repository CRAN
Date/Publication 2021-02-02 00:10:23 UTC
Author Victor Picheny [aut, cre],
    David Ginsbourger Green [aut],
    Olivier Roustant [aut],
    Mickael Binois [ctb],
    Sebastien Marmin [ctb],
    Tobias Wagner [ctb]
Maintainer Victor Picheny <victor.picheny@toulouse.inra.fr>
Encoding UTF-8

R topics documented:

  DiceOptim-package ........................................................... 2
  AEI ................................................................. 8
  AEI.grad ........................................................... 10
DiceOptim-package

Kriging-based optimization methods for computer experiments

Description

Sequential and parallel Kriging-based optimization methods relying on expected improvement criteria.
Details
Note

This work is a follow-up of DiceOptim 1.0, which was produced within the frame of the DICE (Deep Inside Computer Experiments) Consortium between ARMINES, Renault, EDF, IRSN, ONERA and TOTAL S.A.

The authors would like to thank Yves Deville for his precious advice in R programming and packaging, as well as the DICE members for useful feedbacks, and especially Yann Richet (IRSN) for numerous discussions concerning the user-friendliness of this package.

Package rgenoud >=5.3.3. is recommended.

Important functions or methods:

- **EGO.nsteps**: Standard Efficient Global Optimization algorithm with a fixed number of iterations (nsteps) —with model updates including re-estimation of covariance hyperparameters
- **EI**: Expected Improvement criterion (single infill point, noise-free, constraint free problems)
- **max_EI**: Maximization of the EI criterion. No need to specify any objective function
- **qEI.nsteps**: EGO algorithm with batch-sequential (parallel) infill strategy
- **noisy.optimizer**: EGO algorithm for noisy objective functions
- **EGO.cst**: EGO algorithm for (non-linear) constrained problems
- **easyEGO.cst**: User-friendly wrapper for EGO.cst

Author(s)

Victor Picheny (INRA, Castanet-Tolosan, France)
David Ginsbourger (Idiap Research Institute and University of Bern, Switzerland)
Olivier Roustant (Mines Saint-Etienne, France).

with contributions by M. Binois, C. Chevalier, S. Marmin and T. Wagner

References


**Examples**

```r
set.seed(123)

# a 9-points factorial design, and the corresponding response
d <- 2
n <- 9
design.fact <- expand.grid(seq(0,1,length=3), seq(0,1,length=3))
names(design.fact)<-c("x1", "x2")
design.fact <- data.frame(design.fact)
names(design.fact)<-c("x1", "x2")
response.branin <- data.frame(apply(design.fact, 1, branin))
names(response.branin) <- "y"

# model identification
fitted.model1 <- km(~1, design=design.fact, response=response.branin, covtype="gauss", control=list(pop.size=50,trace=FALSE), parinit=c(0.5, 0.5))

### EGO, 5 steps ###############
library(rgenoud)
```

---

The examples above illustrate how to use DiceOptim to perform global optimization on a 9-points factorial design. The `km` function is used for model identification, and the `ego` function is used for optimization with EGO. The `rgenoud` package is used for genetic optimization using derivatives.
nsteps <- 5
ger <- rep(0,d)
gt <- rep(1,d)
EGO <- EGO.nsteps(model=fitted.model1, fun=branin, nsteps=nsteps, 
gt=lower, upper=gt, control=list(pop.size=20, BFGSburnin=2))
print(EGO$par)
print(EGO$value)

# graphics
n.grid <- 15
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
response.grid <- apply(design.grid, 1, branin)
z.grid <- matrix(response.grid, n.grid, n.grid)
contour(x.grid, y.grid, z.grid, 40)

title("EGO")
points(design.fact[,1], design.fact[,2], pch=17, col="blue")
points(EGO$par, pch=19, col="red")
text(EGO$par[,1], EGO$par[,2], labels=1:nsteps, pos=3)

#### Parallel EGO, 3 steps with batches of 3 #######
nsteps <- 3
lower <- rep(0,d)
gt <- rep(1,d)
npoints <- 3 # The batchsize
EGO <- qEGO.nsteps(model = fitted.model1, branin, npoints = npoints, nsteps = nsteps, 
crit="exact", lower, upper, optimcontrol = NULL)
print(EGO$par)
print(EGO$value)

# graphics
contour(x.grid, y.grid, z.grid, 40)

title("qEGO")
points(design.fact[,1], design.fact[,2], pch=17, col="blue")
points(EGO$par, pch=19, col="red")
text(EGO$par[,1], EGO$par[,2], labels=c(tcrossprod(rep(1,npoints),1:nsteps)), pos=3)

##########################################################################
### 2D OPTIMIZATION, NOISY OBJECTIVE ###
##########################################################################
set.seed(10)
library(DiceDesign)
# Set test problem parameters
doe.size <- 9
dim <- 2
test.function <- get("branin2")
lower <- rep(0,1,dim)
gt <- rep(1,1,dim)
nnoise.var <- 0.1

# Build noisy simulator
funnoise <- function(x)
{  
  f.new <- test.function(x) + sqrt(noise.var)*rnorm(n=1)
  return(f.new)
}

# Generate DOE and response

doe <- as.data.frame(lhsDesign(doe.size, dim)$design)
y.tilde <- funnoise(doe)

# Create kriging model
model <- km(y~1, design=doe, response=data.frame(y=y.tilde),
  covtype="gauss", noise.var=rep(noise.var,1,doe.size),
  lower=rep(.1,dim), upper=rep(1,dim), control=list(trace=FALSE))

# Optimisation with noisy.optimizer
optim.param <- list()
optim.param$quantile <- .7
optim.result <- noisy.optimizer(optim.crit="EQI", optim.param=optim.param, model=model,
  n.ite=5, noise.var=noise.var, funnoise=funnoise, lower=lower, upper=upper,
  NoiseReEstimate=FALSE, CovReEstimate=FALSE)

print(optim.result$best.x)

##########################################################################
### 2D OPTIMIZATION, 2 INEQUALITY CONSTRAINTS ###
##########################################################################

set.seed(25468)
library(DiceDesign)

fun <- goldsteinprice
fun1.cst <- function(x){return(-branin(x) + 25)}
constraint <- function(x){return(c(fun1.cst(x), fun2.cst(x)))}

lower <- rep(0, 2)
upper <- rep(1, 2)

## Optimization using the Expected Feasible Improvement criterion
res <- easyEGO.cst(fun=fun, constraint=constraint, n.cst=2, lower=lower, upper=upper, budget=10,
  control=list(method="EFI", inneroptim="genoud", maxit=20))

cat("best design found: ", res$par, "\n")
cat("corresponding objective and constraints: ", res$value, "\n")

# Objective function in colour, constraint boundaries in red
# Initial DoE: white circles, added points: blue crosses, best solution: red cross

n.grid <- 15
test.grid <- expand.grid(X1 = seq(0, 1, length.out = n.grid), X2 = seq(0, 1, length.out = n.grid))
obj.grid <- apply(test.grid, 1, fun)
cst1.grid <- apply(test.grid, 1, fun1.cst)
cst2.grid <- apply(test.grid, 1, fun2.cst)
filled.contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), nlevels = 50,
  matrix(obj.grid, n.grid), main = "Two inequality constraints",
  xlab = expression(x[1]), ylab = expression(x[2]), color = terrain.colors,
plot.axes = (axis(1); axis(2);
    contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid),
          matrix(cst1.grid, n.grid), level = 0, add=TRUE,
          draw.labels=FALSE, lwd=1.5, col = "red"
    )
    contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid),
          matrix(cst2.grid, n.grid), level = 0, add=TRUE, draw.labels=FALSE,
          lwd=1.5, col = "red"
    )
    points(res$history$X, col = "blue", pch = 4, lwd = 2)
    points(res$par[1], res$par[2], col = "red", pch = 4, lwd = 2, cex=2)
)

AEI

Augmented Expected Improvement

Description

Evaluation of the Augmented Expected Improvement (AEI) criterion, which is a modification of the classical EI criterion for noisy functions. The AEI consists of the regular EI multiplied by a penalization function that accounts for the diminishing payoff of observation replicates. The current minimum y.min is chosen as the kriging predictor of the observation with smallest kriging quantile.

Usage

AEI(x, model, new.noise.var = 0, y.min = NULL, type = "UK", envir = NULL)

Arguments

x the input vector at which one wants to evaluate the criterion
model a Kriging model of "km" class
new.noise.var the (scalar) noise variance of the future observation.
y.min The kriging predictor at the current best point (point with smallest kriging quantile). If not provided, this quantity is evaluated.
type Kriging type: "SK" or "UK"
envir environment for saving intermediate calculations and reusing them within AEI.grad

Value

Augmented Expected Improvement

Author(s)

Victor Picheny
David Ginsbourger


References


Examples

#########################################################################
### AEI SURFACE ASSOCIATED WITH AN ORDINARY KRIGING MODEL ####
### OF THE BRANIN FUNCTION KNOWN AT A 12-POINT LATIN HYPERCUBE DESIGN ####
#########################################################################

currentSeed <- get.seed()
set.seed(421)

# Set test problem parameters
doe.size <- 12
dim <- 2
test.function <- get("branin2")
lower <- rep(0,1,dim)
upper <- rep(1,1,dim)
noise.var <- 0.2

# Generate DOE and response
doe <- as.data.frame(matrix(runif(doe.size*dim),doe.size))
y.tilde <- rep(0, 1, doe.size)
for (i in 1:doe.size) {
y.tilde[i] <- test.function(doe[i,]) + sqrt(noise.var)*rnorm(n=1)
}
y.tilde <- as.numeric(y.tilde)

# Create kriging model
model <- km(y~1, design=doe, response=data.frame(y=y.tilde),
            covtype="gauss", noise.var=rep(noise.var,1,doe.size),
            lower=rep(.1,dim), upper=rep(1,dim), control=list(trace=FALSE))

# Compute actual function and criterion on a grid
n.grid <- 12 # Change to 21 for a nicer picture
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
nt <- nrow(design.grid)
crit.grid <- rep(0,1,nt)
func.grid <- rep(0,1,nt)

crit.grid <- apply(design.grid, 1, AEI, model=model, new.noise.var=noise.var)
func.grid <- apply(design.grid, 1, test.function)

# Compute kriging mean and variance on a grid
names(design.grid) <- c("V1","V2")
pred <- predict.km(model, newdata=design.grid, type="UK")
mk.grid <- pred$m
sk.grid <- pred$sd
# Plot actual function
z.grid <- matrix(func.grid, n.grid, n.grid)
filled.contour(x.grid, y.grid, z.grid, nlevels=50, color = rainbow, plot.axes = {title("Actual function"); points(model@X[,1],model@X[,2],pch=17,col="blue"); axis(1); axis(2))}

# Plot Kriging mean
z.grid <- matrix(mk.grid, n.grid, n.grid)
filled.contour(x.grid, y.grid, z.grid, nlevels=50, color = rainbow, plot.axes = {title("Kriging mean"); points(model@X[,1],model@X[,2],pch=17,col="blue"); axis(1); axis(2))}

# Plot Kriging variance
z.grid <- matrix(sk.grid^2, n.grid, n.grid)
filled.contour(x.grid, y.grid, z.grid, nlevels=50, color = rainbow, plot.axes = {title("Kriging variance"); points(model@X[,1],model@X[,2],pch=17,col="blue"); axis(1); axis(2))}

# Plot AEI criterion
z.grid <- matrix(crit.grid, n.grid, n.grid)
filled.contour(x.grid, y.grid, z.grid, nlevels=50, color = rainbow, plot.axes = {title("AEI"); points(model@X[,1],model@X[,2],pch=17,col="blue"); axis(1); axis(2))}

---

**AEI.grad**

**AEI's Gradient**

**Description**

Analytical gradient of the Augmented Expected Improvement (AEI) criterion.

**Usage**

`AEI.grad(x, model, new.noise.var = 0, y.min = NULL, type = "UK", envir = NULL)`

**Arguments**

- **x**
  - the input vector at which one wants to evaluate the criterion
- **model**
  - a Kriging model of "km" class
- **new.noise.var**
  - the (scalar) noise variance of the new observation.
- **y.min**
  - The kriging predictor at the current best point (point with smallest kriging quantile). If not provided, this quantity is evaluated.
- **type**
  - Kriging type: "SK" or "UK"
- **envir**
  - environment for inheriting intermediate calculations from AEI
Value

Gradient of the Augmented Expected Improvement

Author(s)

Victor Picheny
David Ginsbourger

Examples

```r
set.seed(421)

# Set test problem parameters
doe.size <- 12
dim <- 2
test.function <- get("branin2")
lower <- rep(0,1,dim)
upper <- rep(1,1,dim)
noise.var <- 0.2

# Generate DOE and response
doe <- as.data.frame(matrix(runif(doe.size*dim),doe.size))
y.tilde <- rep(0, 1, doe.size)
for (i in 1:doe.size) {
y.tilde[i] <- test.function(doe[i,]) + sqrt(noise.var)*rnorm(n=1)
}
y.tilde <- as.numeric(y.tilde)

# Create kriging model
model <- km(y~1, design=doe, response=data.frame(y=y.tilde),
covtype="gauss", noise.var=rep(noise.var,1,doe.size),
lower=rep(.1,dim), upper=rep(1,dim), control=list(trace=FALSE))

# Compute actual function and criterion on a grid
n.grid <- 8 # Change to 21 for a nicer picture
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
nt <- nrow(design.grid)
crit.grid <- apply(design.grid, 1, AEI, model=model, new.noise.var=noise.var)
crit.grad <- t(apply(design.grid, 1, AEI.grad, model=model, new.noise.var=noise.var))
z.grid <- matrix(crit.grid, n.grid, n.grid)
contour(x.grid,y.grid, z.grid, 30)
title("AEI and its gradient")
points(model@X[,1],model@X[,2],pch=17,col="blue")

for (i in 1:nt) {
x <- design.grid[i,]
suppressWarnings(arrows(x$Var1,x$Var2, x$Var1+crit.grad[i,1]*.6,x$Var2+crit.grad[i,2]*.6,
```
**AKG**

Approximate Knowledge Gradient (AKG)

---

**Description**

Evaluation of the Approximate Knowledge Gradient (AKG) criterion.

**Usage**

```r
AKG(x, model, new.noise.var = 0, type = "UK", envir = NULL)
```

**Arguments**

- `x`: the input vector at which one wants to evaluate the criterion
- `model`: a Kriging model of "km" class
- `new.noise.var`: (scalar) noise variance of the future observation. Default value is 0 (noise-free observation).
- `type`: Kriging type: "SK" or "UK"
- `envir`: environment for saving intermediate calculations and reusing them within AKG.grad

**Value**

Approximate Knowledge Gradient

**Author(s)**

Victor Picheny

David Ginsbourger

**References**

Examples

#########################################################################
### AKG SURFACE ASSOCIATED WITH AN ORDINARY KRIGING MODEL ####
### OF THE BRANIN FUNCTION KNOWN AT A 12-POINT LATIN HYPERCUBE DESIGN ####
#########################################################################

set.seed(421)
# Set test problem parameters
doe.size <- 12
dim <- 2
test.function <- get("branin2")
lower <- rep(0,1,dim)
upper <- rep(1,1,dim)
noise.var <- 0.2

# Generate DOE and response
doe <- as.data.frame(matrix(runif(doe.size*dim),doe.size))
y.tilde <- rep(0, 1, doe.size)
for (i in 1:doe.size) {
  y.tilde[i] <- test.function(doe[i,]) + sqrt(noise.var)*rnorm(n=1)
}
y.tilde <- as.numeric(y.tilde)

# Create kriging model
model <- km(y~1, design=doe, response=data.frame(y=y.tilde),
covtype="gauss", noise.var=rep(noise.var,1,doe.size),
lower=rep(.1,dim), upper=rep(1,dim), control=list(trace=FALSE))

# Compute actual function and criterion on a grid
n.grid <- 12 # Change to 21 for a nicer picture
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
nt <- nrow(design.grid)
crit.grid <- apply(design.grid, 1, AKG, model=model, new.noise.var=noise.var)
func.grid <- apply(design.grid, 1, test.function)

# Compute kriging mean and variance on a grid
names(design.grid) <- c("V1","V2")
pred <- predict.km(model, newdata=design.grid, type="UK")
mk.grid <- pred$m
sk.grid <- pred$sd

# Plot actual function
z.grid <- matrix(func.grid, n.grid, n.grid)
filled.contour(x.grid,y.grid, z.grid, nlevels=50, color = topo.colors,
plot.axes = (title("Actual function");
  points(model@X[,1],model@X[,2],pch=17,col="blue");
  axis(1); axis(2)))

# Plot Kriging mean
z.grid <- matrix(mk.grid, n.grid, n.grid)
AKG.grad

AKG's Gradient

Description
Gradient of the Approximate Knowledge Gradient (AKG) criterion.

Usage
AKG.grad(x, model, new.noise.var = 0, type = "UK", envir = NULL)

Arguments
- **x**  the input vector at which one wants to evaluate the criterion
- **model**  a Kriging model of "km" class
- **new.noise.var**  (scalar) noise variance of the future observation. Default value is 0 (noise-free observation).
- **type**  Kriging type: "SK" or "UK"
- **envir**  optional: environment for reusing intermediate calculations from AKG

Value
Gradient of the Approximate Knowledge Gradient

Author(s)
Victor Picheny
Examples

##########################################################################
### AKG SURFACE AND ITS GRADIENT ASSOCIATED WITH AN ORDINARY          
### KRIGING MODEL                                                     
### OF THE BRANIN FUNCTION KNOWN AT A 12-POINT LATIN HYPERCUBE DESIGN ###
##########################################################################

set.seed(421)

# Set test problem parameters
doe.size <- 12
dim <- 2
test.function <- get("branin2")
lower <- rep(0,1,dim)
upper <- rep(1,1,dim)
noise.var <- 0.2

# Generate DOE and response
doe <- as.data.frame(matrix(runif(doe.size*dim),doe.size))
y.tilde <- rep(0, 1, doe.size)
for (i in 1:doe.size) {
y.tilde[i] <- test.function(doe[i,]) + sqrt(noise.var)*rnorm(n=1)
}
y.tilde <- as.numeric(y.tilde)

# Create kriging model
model <- km(y~1, design=doe, response=data.frame(y=y.tilde),
covtype="gauss", noise.var=rep(noise.var,1,doe.size),
lower=rep(.1,dim), upper=rep(1,dim), control=list(trace=FALSE))

# Compute actual function and criterion on a grid
n.grid <- 9 # Change to 21 for a nicer picture
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
nt <- nrow(design.grid)
crit.grid <- apply(design.grid, 1, AKG, model=model, new.noise.var=noise.var)
crit.grad <- t(apply(design.grid, 1, AKG.grad, model=model, new.noise.var=noise.var))
z.grid <- matrix(crit.grid, n.grid, n.grid)
contour(x.grid,y.grid, z.grid, 30)
title("AKG and its gradient")
points(model@X[,1],model@X[,2],pch=17,col="blue")

for (i in 1:nt) {
x <- design.grid[i,]
suppressWarnings(arrows(x$Var1,x$Var2, x$Var1+crit.grad[i,1]*.2,x$Var2+crit.grad[i,2]*.2,
length=0.04,code=2,col="orange",lwd=2))
}
checkPredict  Prevention of numerical instability for a new observation

Description

Check that the new point is not too close to already known observations to avoid numerical issues. Closeness can be estimated with several distances.

Usage

checkPredict(x, model, threshold = 1e-04, distance = "covdist", type = "UK")

Arguments

x a vector representing the input to check,
model list of objects of class km, one for each objective functions,
threshold optional value for the minimal distance to an existing observation, default to 1e-4,
distance selection of the distance between new observations, between "euclidean", "covdist" (default) and "covratio", see details,
type "SK" or "UK" (default), depending whether uncertainty related to trend estimation has to be taken into account.

Details

If the distance between x and the closest observations in model is below threshold, x should not be evaluated to avoid numerical instabilities. The distance can simply be the Euclidean distance or the canonical distance associated with the kriging covariance k:

\[ d(x, y) = \sqrt{k(x, x) - 2k(x, y) + k(y, y)}. \]

The last solution is the ratio between the prediction variance at \( x \) and the variance of the process.

Value

TRUE if the point should not be tested.

Author(s)

Mickael Binois
critcst_optimizer

Maximization of constrained Expected Improvement criteria

Description

Given objects of class km for the objective and constraints, and a set of tuning parameters (lower, upper and critcontrol), critcst_optimizer performs the maximization of a constrained Expected Improvement or SUR criterion and delivers the next point to be visited in an EGO-like procedure.

The latter maximization relies either on a genetic algorithm using derivatives, genoud or exhaustive search at pre-specified points. It is important to remark that the information needed about the objective and constraint functions reduces here to the vector of response values embedded in the models (no call to the objective/constraint functions or simulators (except possibly for the objective)).

Usage

critcst_optimizer(
  crit = "EFI",
  model.fun,
  model.constraint,
  equality = FALSE,
  lower,
  upper,
  type = "UK",
  critcontrol = NULL,
  optimcontrol = NULL
)

Arguments

crit sampling criterion. Three choices are available: "EFI", "AL" and "SUR",
model.fun object of class km corresponding to the objective function, or, if the objective function is fast-to-evaluate, either the objective function to be minimized or a fastfun object, see details and examples below,
model.constraint either one or a list of models of class km, one for each constraint,
equality either FALSE if all constraints are inequalities, or a Boolean vector indicating which are equalities,
lower vector of lower bounds for the variables to be optimized over,
upper vector of upper bounds for the variables to be optimized over,
type "SK" or "UK" (default), depending whether uncertainty related to trend estimation has to be taken into account.
critcontrol optional list of control parameters for criterion crit, see details. Options for the checkPredict function: threshold (1e-4) and distance (covdist) are used to avoid numerical issues occurring when adding points too close to the existing ones.

optimcontrol optional list of control parameters for optimization of the selected infill criterion. "method" set the optimization method; one can choose between "discrete" and "genoud". For each method, further parameters can be set. For "discrete", one has to provide the argument "candidate.points". For "genoud", one can control, among others, "pop.size" (default: \([N = 3*2^\text{dim} \text{ for dim} < 6 \text{ and } N = 32*\text{dim otherwise}]\), "max.generations" (12), "wait.generations" (2)), see genoud. Numbers into brackets are the default values. @return A list with components:

- par: The best set of parameters found,
- value: The value of expected improvement at par.

Details

Extension of the function max_EI for constrained optimization.

Available infill criteria with crit are:

- Expected Probability of Feasibility (EFI) critEFI,
- Augmented Lagrangian (AL) critAL,
- Stepwise Uncertainty Reduction of the excursion volume (SUR) critSURcst.

Depending on the selected criterion, parameters can be given with critcontrol. Also options for checkPredict are available. More precisions are given in the corresponding help pages.

If the objective function to minimize is inexpensive, i.e. no need of a kriging model, then one can provide it in model.obj, which is handled next with class fastfun (or directly as a fastfun object). See example below.

In the case of equality constraints, it is possible to define them with equality. Additionally, one can modify the tolerance on the constraints using the tolConstraints component of critcontrol: an optional vector giving a tolerance for each of the constraints (equality or inequality). It is highly recommended to use it when there are equality constraints since the default tolerance of 0.05 (resp. 0 for inequality constraints) in such case might not be suited.

Author(s)

Victor Picheny
Mickael Binois
References


See Also

`critcst_optimizer`, `crit_EFI`, `crit_AL`, `crit_SUR_cst`

Examples

```r
#---------------------------------------------------------------------------
# 2D objective function, 2 cases
#---------------------------------------------------------------------------

set.seed(2546)
library(DiceDesign)

n_var <- 2
fun <- branin

fun1.cst <- function(x){return(goldsteinprice(x)+.5)}

# Constraint function with vectorial output
cstfun <- function(x){return(c(fun1.cst(x), fun2.cst(x)))}

n.grid <- 31
test.grid <- expand.grid(X1 = seq(0, 1, length.out = n.grid), X2 = seq(0, 1, length.out = n.grid))
obj.grid <- apply(test.grid, 1, fun)
cst1.grid <- apply(test.grid, 1, fun1.cst)
cst2.grid <- apply(test.grid, 1, fun2.cst)

n_appr <- 12
design.grid <- round(maximinESE_LHS(lhsDesign(n_appr, n_var, seed = 2)$design)$design, 1)
obj.init <- apply(design.grid, 1, fun)
cst1.init <- apply(design.grid, 1, fun1.cst)
cst2.init <- apply(design.grid, 1, fun2.cst)
model.fun <- km(~., design = design.grid, response = obj.init)
model.constraint1 <- km(~., design = design.grid, response = cst1.init, lower=c(.2,.2))
model.constraint2 <- km(~., design = design.grid, response = cst2.init, lower=c(.2,.2))
models.cst <- list(model.constraint1, model.constraint2)

lower <- rep(0, n_var)
upper <- rep(1, n_var)

# Augmented Lagrangian Improvement, fast objective function, two ineq constraints,
# optimization with genoud
```
#--------------------------
critcontrol <- list(lambda=c(.5, 2), rho=.5)
optimcontrol <- list(method = "genoud", max.generations=10, pop.size=20)

AL_grid <- apply(test.grid, 1, crit_AL, model.fun = fastfun(fun, design.grid),
                   model.constraint = models.cst, critcontrol=critcontrol)
cstEGO1 <- critcst_optimizer(crit = "AL", model.fun = fun,
                              model.constraint = models.cst, equality = FALSE,
                              lower = lower, upper = upper,
                              optimcontrol = optimcontrol, critcontrol=critcontrol)

filled.contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), nlevels = 50,
               matrix(AL_grid, n.grid), main = "AL map and its maximizer (blue)",
               xlab = expression(x[1]), ylab = expression(x[2]), color = terrain.colors,
               plot.axes = {axis(1); axis(2);
                           points(design.grid[,1], design.grid[,2], pch = 21, bg = "white")
                           contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid),
                                   matrix(obj.grid, n.grid), nlevels = 10, add=TRUE, drawlabels=TRUE,
                                   col = "black")
                           contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid),
                                   matrix(cst1.grid, n.grid), level = 0, add=TRUE, drawlabels=FALSE,
                                   lwd=1.5, col = "red")
                           contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid),
                                   matrix(cst2.grid, n.grid), level = 0, add=TRUE, drawlabels=FALSE,
                                   lwd=1.5, col = "red")
                           points(cstEGO1$par, col = "blue", pch = 4, lwd = 2)
                }

#--------------------------
# SUR, expensive objective function, one equality constraint,
# optimization with genoud, integration on a regular grid
#--------------------------
optimcontrol <- list(method = "genoud", s = 40, maxit = 40)
critcontrol <- list(tolConstraints = .15, integration.points=as.matrix(test.grid))

SUR_grid <- apply(test.grid, 1, crit_SUR_cst, model.fun = model.fun,
                   model.constraint = model.constraint1, equality = TRUE, critcontrol = critcontrol)
cstEGO2 <- critcst_optimizer(crit = "SUR", model.fun = model.fun,
                              model.constraint = model.constraint1, equality = TRUE,
                              lower = lower, upper = upper,
                              optimcontrol = optimcontrol, critcontrol = critcontrol)

filled.contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), nlevels = 50,
               matrix(SUR_grid, n.grid), main = "SUR map and its maximizer (blue)",
               xlab = expression(x[1]), ylab = expression(x[2]), color = terrain.colors,
               plot.axes = {axis(1); axis(2);
                           points(design.grid[,1], design.grid[,2], pch = 21, bg = "white")
                           contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid),
                                   matrix(obj.grid, n.grid), nlevels = 10, add=TRUE,
                                   drawlabels=TRUE, col = "black")
                           contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid),
                                   matrix(cst1.grid, n.grid), level = 0, add=TRUE,
                                   drawlabels=FALSE, col = "red")
                           contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid),
                                   matrix(cst2.grid, n.grid), level = 0, add=TRUE,
                                   drawlabels=FALSE, col = "red")
                           points(cstEGO2$par, col = "blue", pch = 4, lwd = 2)\n                )
crit_AL

matrix(cst1.grid, n.grid), level = c(-critcontrol$tolConstraints, 
critcontrol$tolConstraints),
add=TRUE, drawlabels=FALSE,lwd=1.5, col = "orange")
points(cstEGO2$par, col = "blue", pch = 4, lwd = 2)
}

---

crit_AL  Expected Augmented Lagrangian Improvement

Description

Computes the Expected Augmented Lagrangian Improvement at current location, with or without slack variables. Depending on the cases, the computation is either analytical (very fast), based on MC integration (slow) or on the CDF of a weighted sum of non-central chi-square (WNCS) variates (intermediate).

Usage

crit_AL(
  x,
  model.fun,
  model.constraint,
  equality = FALSE,
  critcontrol = NULL,
  type = "UK"
)

Arguments

x     either a vector representing the design or the design AND slack variables (see details)
model.fun object of class km correspostnding to the objective function, or, if the objective function is fast-to-evaluate, a fastfun object,
model.constraint either one or a list of objects of class km, one for each constraint function,
equality    either FALSE if all constraints are for inequalities, or a vector of Booleans indicating which are equalities
critcontrol optional list with the following arguments:
  • slack: logical. If TRUE, slack variables are used for inequality constraints (see Details)
  • rho: penalty term (scalar),
  • lambda: Lagrange multipliers (vector of size the number of constraints),
  • elit: logical. If TRUE, sets the criterion to zero for all x’s not improving the objective function
• `n.mc`: number of Monte-Carlo drawings used to evaluate the criterion (see Details)
• `nt`: number of discretization points for the WNCS distribution (see Details)
• `tolConstraints`, an optional vector giving a tolerance (> 0) for each of the constraints (equality or inequality). It is highly recommended to use it when there are equality constraints since the default tolerance of 0.05 in such case might not be suited.

Options for the `checkPredict` function: `threshold(1e-4)` and `distance(covdist)` are used to avoid numerical issues occurring when adding points too close to the existing ones.

`type` "SK" or "UK" (by default), depending whether uncertainty related to trend estimation has to be taken into account.

Details

The AL can be used with or without the help of slack variables for the inequality constraints. If `critcontrol$slack=FALSE`: With a single constraint (inequality or equality) and a fast objective, a very fast formula is used to compute the criterion (recommended setting). Otherwise, an MC estimator of the criterion is used, which is much more costly. The argument `critcontrol$n.mc` tunes the precision of the estimator. On both cases `x` must be of size `d`.

If `critcontrol$slack=TRUE`: Slack variables are used to handle the inequality constraints. They can be provided directly through `x`, which should be of size `d+` the number of inequality constraints. The last values of `x` are slack variables scaled to [0,1].

If `x` is of size `d`, estimates of optimal slack variable are used.

Value

The Expected Augmented Lagrangian Improvement at `x`.

Author(s)

Victor Picheny
Mickael Binois

References


See Also

EI from package DiceOptim, crit_EFI, crit_SUR_cst.
Examples

#---------------------------------------------------------------------------
# Expected Augmented Lagrangian Improvement surface with one inequality constraint,
# fast objective
#---------------------------------------------------------------------------

set.seed(25468)
library(DiceDesign)

n_var <- 2
fun.obj <- goldsteinprice
fun.cst <- function(x){return(-branin(x) + 25)}
n.grid <- 31
test.grid <- expand.grid(X1 = seq(0, 1, length.out = n.grid), X2 = seq(0, 1, length.out = n.grid))
obj.grid <- apply(test.grid, 1, fun.obj)
cst.grid <- apply(test.grid, 1, fun.cst)
n.init <- 15
design.grid <- round(maximinESE_LHS(lhsDesign(n.init, n_var, seed = 42)$design)$design, 1)
obj.init <- apply(design.grid, 1, fun.obj)
cst.init <- apply(design.grid, 1, fun.cst)
model.constraint <- km(~., design = design.grid, response = cst.init)
model.fun <- fastfun(fun.obj, design.grid)
AL_grid <- apply(test.grid, 1, crit_AL, model.fun = model.fun,
model.constraint = model.constraint)

filled.contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), nlevels = 50,
matrix(AL_grid, n.grid), main = "Expected AL Improvement",
xlab = expression(x[1]), ylab = expression(x[2]), color = terrain.colors,
plot.axes = {axis(1); axis(2); points(design.grid[,1], design.grid[,2], pch = 21, bg = "white")
contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid),
matrix(obj.grid, n.grid), nlevels = 10,
add=TRUE, drawlabels=TRUE, col = "black")
contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid),
matrix(cst.grid, n.grid), level = 0, add=TRUE,
drawlabels=FALSE,lwd=1.5, col = "red")
}
}

#---------------------------------------------------------------------------
# Expected AL Improvement surface with one inequality and one equality constraint,
# using slack variables
#---------------------------------------------------------------------------

set.seed(25468)
library(DiceDesign)

n_var <- 2
fun.obj <- goldsteinprice
fun.csteq <- function(x){return(branin(x) - 25)}
n.grid <- 51
test.grid <- expand.grid(X1 = seq(0, 1, length.out = n.grid), X2 = seq(0, 1, length.out = n.grid))
obj.grid <- apply(test.grid, 1, fun.obj)
cstineq.grid <- apply(test.grid, 1, fun.cstineq)
csteq.grid <- apply(test.grid, 1, fun.csteq)
n.init <- 25
design.grid <- round(maximinESE_LHS(lhsDesign(n.init, n_var, seed = 42)$design)$design, 1)
obj.init <- apply(design.grid, 1, fun.obj)
cstineq.init <- apply(design.grid, 1, fun.cstineq)
csteq.init <- apply(design.grid, 1, fun.csteq)
model.fun <- km(~., design = design.grid, response = obj.init)
model.constraintineq <- km(~., design = design.grid, response = cstineq.init)
model.constrainteq <- km(~., design = design.grid, response = csteq.init)
models.cst <- list(model.constraintineq, model.constrainteq)
AL_grid <- apply(test.grid, 1, crit_AL, model.fun = model.fun, model.constraint = models.cst,
                  equality = c(FALSE, TRUE), critcontrol = list(tolConstraints = c(0.05, 3), slack=TRUE))

crit_EFI

Expected Feasible Improvement

Description

Computes the Expected Feasible Improvement at current location. The current feasible minimum
of the observations can be replaced by an arbitrary value (plugin), which is useful in particular in
noisy frameworks.

Usage

crit_EFI(
  x,
  ...)
crit_EFI

model.fun,
model.constraint,
equality = FALSE,
critcontrol = NULL,
plugin = NULL,
type = "UK"
)

Arguments

x a vector representing the input for which one wishes to calculate EFI,
model.fun object of class km corresponding to the objective function, or, if the objective function is fast-to-evaluate, a fastfun object,
model.constraint either one or a list of objects of class km, one for each constraint function,
equality either FALSE if all constraints are for inequalities, else a vector of boolean indicating which are equalities,
critcontrol optional list with argument tolConstraints, an optional vector giving a tolerance (> 0) for each of the constraints (equality or inequality). It is highly recommended to use it when there are equality constraints since the default tolerance of 0.05 in such case might not be suited.

Options for the checkPredict function: threshold(1e-4) and distance(covdist) are used to avoid numerical issues occurring when adding points too close to the existing ones.

plugin optional scalar: if provided, it replaces the feasible minimum of the current observations. If set to Inf, e.g. when there is no feasible solution, then the criterion is equal to the probability of feasibility,
type "SK" or "UK" (by default), depending whether uncertainty related to trend estimation has to be taken into account.

Value

The Expected Feasible Improvement at x.

Author(s)

Victor Picheny
Mickael Binois

References

See Also

EI from package DiceOptim, crit_AL, crit_SUR_cst.

Examples

```r
#---------------------------------------------------------------------------
# Expected Feasible Improvement surface with one inequality constraint
#---------------------------------------------------------------------------

set.seed(25468)
library(DiceDesign)

n_var <- 2
fun.obj <- goldsteinprice
fun.cst <- function(x){return(-branin(x) + 25)}
n.grid <- 51

test.grid <- expand.grid(X1 = seq(0, 1, length.out = n.grid), X2 = seq(0, 1, length.out = n.grid))
obj.grid <- apply(test.grid, 1, fun.obj)
cst.grid <- apply(test.grid, 1, fun.cst)
n.init <- 15
design.grid <- round(maximinESE_LHS(lhsDesign(n.init, n_var, seed = 42)$design)$design, 1)
obj.init <- apply(design.grid, 1, fun.obj)
cst.init <- apply(design.grid, 1, fun.cst)
model.fun <- km(~., design = design.grid, response = obj.init)
model.constraint <- km(~., design = design.grid, response = cst.init)

EFI_grid <- apply(test.grid, 1, crit_EFI, model.fun = model.fun, model.constraint = model.constraint)

filled.contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), nlevels = 50, matrix(EFI_grid, n.grid), main = "Expected Feasible Improvement", xlab = expression(x[1]), ylab = expression(x[2]), color = terrain.colors, plot.axes = {axis(1); axis(2);
points(design.grid[,1], design.grid[,2], pch = 21, bg = "white")
contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), matrix(obj.grid, n.grid), nlevels = 10, add=TRUE,drawlabels=TRUE, col = "black")
contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), matrix(cst.grid, n.grid), level = 0, add=TRUE, drawlabels=FALSE,lwd=1.5, col = "red")
})

#---------------------------------------------------------------------------
# Expected Feasible Improvement surface with one inequality and one equality constraint
#---------------------------------------------------------------------------

set.seed(25468)
library(DiceDesign)

n_var <- 2
fun.obj <- goldsteinprice
```
fun.csteq <- function(x){return(branin(x) - 25)}
n.grid <- 51
test.grid <- expand.grid(X1 = seq(0, 1, length.out = n.grid), X2 = seq(0, 1, length.out = n.grid))
obj.grid <- apply(test.grid, 1, fun.obj)
cstineq.grid <- apply(test.grid, 1, fun.cstineq)
csteq.grid <- apply(test.grid, 1, fun.csteq)
n.init <- 25
design.grid <- round(maximinESE_LHS(lhsDesign(n.init, n_var, seed = 42)$design)$design, 1)
obj.init <- apply(design.grid, 1, fun.obj)
cstineq.init <- apply(design.grid, 1, fun.cstineq)
csteq.init <- apply(design.grid, 1, fun.csteq)
model.fun <- km(~., design = design.grid, response = obj.init)
model.constraintineq <- km(~., design = design.grid, response = cstineq.init)
model.constrainteq <- km(~., design = design.grid, response = csteq.init)
models.cst <- list(model.constraintineq, model.constrainteq)
 EFI_grid <- apply(test.grid, 1, crit_EFI, model.fun = model.fun, model.constraint = models.cst, equality = c(FALSE, TRUE), critcontrol = list(tolConstraints = c(0.05, 3)))

filled.contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), nlevels = 50, matrix(EFI_grid, n.grid), main = "Expected Feasible Improvement", xlab = expression(x[1]), ylab = expression(x[2]), color = terrain.colors, plot.axes = {axis(1); axis(2); points(design.grid[,1], design.grid[,2], pch = 21, bg = "white")
contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), matrix(obj.grid, n.grid), nlevels = 10, add=TRUE, drawlabels=TRUE, col = "black")
contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), matrix(cstineq.grid, n.grid), level = 0, add=TRUE, drawlabels=FALSE, lwd=1.5, col = "red")
contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), matrix(csteq.grid, n.grid), level = 0, add=TRUE, drawlabels=FALSE, lwd=1.5, col = "orange")
})

---

crit_SUR_cst  Stepwise Uncertainty Reduction criterion

Description
Computes the Stepwise Uncertainty Reduction (SUR) criterion at current location

Usage

```R
crit_SUR_cst(x,
```

model.fun, 
model.constraint, 
equality = FALSE, 
critcontrol = NULL, 
type = "UK"
)

Arguments

x an vector representing the input for which one wishes to calculate SUR,
model.fun object of class km corresponding to the objective function, or, if the objective function is fast-to-evaluate, a fastfun object,
model.constraint either one or a list of objects of class km, one for each constraint function,
equality either FALSE if all constraints are for inequalities, else a vector of boolean indicating which are equalities 
critcontrol optional list with arguments:
  • tolConstraints optional vector giving a tolerance (> 0) for each of the constraints (equality or inequality). It is highly recommended to use it when there are equality constraints since the default tolerance of 0.05 in such case might not be suited;
  • integration.points and integration.weights: optional matrix and vector of integration points;
  • precalc.data.cst, precalc.data.obj, mn.X.cst, sn.X.cst, mn.X.obj, sn.X.obj: useful quantities for the fast evaluation of the criterion.
  • Options for the checkPredict function: threshold (1e-4) and distance (covdist) are used to avoid numerical issues occurring when adding points too close to the existing ones.
type "SK" or "UK" (by default), depending whether uncertainty related to trend estimation has to be taken into account.

Value

The Stepwise Uncertainty Reduction criterion at x.

Author(s)

Victor Picheny
Mickael Binois

References

crit_SUR_cst

See Also

EI from package DiceOptim, critEFI, critAL.

Examples

#-------------------------------------------------------------------------------------
# Stepwise Uncertainty Reduction criterion surface with one inequality constraint
#-------------------------------------------------------------------------------------

set.seed(25468)
library(DiceDesign)

n_var <- 2
fun.obj <- goldsteinprice
fun.cst <- function(x){return(-branin(x) + 25)}
n.grid <- 21
test.grid <- expand.grid(X1 = seq(0, 1, length.out = n.grid), X2 = seq(0, 1, length.out = n.grid))
obj.grid <- apply(test.grid, 1, fun.obj)
cst.grid <- apply(test.grid, 1, fun.cst)
n_appr <- 15
design.grid <- round(maximinESE_LHS(lhsDesign(n_appr, n_var, seed = 42)$design)$design, 1)
obj.init <- apply(design.grid, 1, fun.obj)
cst.init <- apply(design.grid, 1, fun.cst)
model.fun <- km(~., design = design.grid, response = obj.init)
model.constraint <- km(~., design = design.grid, response = cst.init)

integration.param <- integration_design_cst(integcontrol = list(integration.points = test.grid),
lower = rep(0, n_var), upper = rep(1, n_var))
SUR_grid <- apply(test.grid, 1, crit_SUR_cst, model.fun = model.fun,
model.constraint = model.constraint, critcontrol=integration.param)
filled.contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), nlevels = 50,
matrix(SUR_grid, n.grid), main = "SUR criterion",
xlab = expression(x[1]), ylab = expression(x[2]), color = terrain.colors,
plot.axes = {axis(1); axis(2);
points(design.grid[,1], design.grid[,2], pch = 21, bg = "white")
contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid),
matrix(obj.grid, n.grid), nlevels = 10,
add=TRUE,drawlabels=TRUE, col = "black")
contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid),
matrix(cst.grid, n.grid), level = 0, add=TRUE,
drawlabels=FALSE,lwd=1.5, col = "red")
})

#-------------------------------------------------------------------------------------
# SUR with one inequality and one equality constraint
#-------------------------------------------------------------------------------------

set.seed(25468)
library(DiceDesign)

n_var <- 2
fun.obj <- goldsteinprice
fun.csteq <- function(x){return(branin(x) - 25)}
n.grid <- 21

test.grid <- expand.grid(X1 = seq(0, 1, length.out = n.grid), X2 = seq(0, 1, length.out = n.grid))
obj.grid <- apply(test.grid, 1, fun.obj)
cstineq.grid <- apply(test.grid, 1, fun.cstineq)
csteq.grid <- apply(test.grid, 1, fun.csteq)
n_appr <- 25
design.grid <- round(maximinESE_LHS(lhsDesign(n_appr, n_var, seed = 42)$design)$design, 1)

obj.init <- apply(design.grid, 1, fun.obj)
cstineq.init <- apply(design.grid, 1, fun.cstineq)
csteq.init <- apply(design.grid, 1, fun.csteq)

model.fun <- km(~., design = design.grid, response = obj.init)
model.constraintineq <- km(~., design = design.grid, response = cstineq.init)
model.constrainteq <- km(~., design = design.grid, response = csteq.init)

models.cst <- list(model.constraintineq, model.constrainteq)

SUR_grid <- apply(test.grid, 1, crit_SUR_cst, model.fun = model.fun, model.constraint = models.cst,
equality = c(FALSE, TRUE), critcontrol = list(tolConstraints = c(0.05, 3), integration.points=integration.param$integration.points))

filled.contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), nlevels = 50,
      matrix(SUR_grid, n.grid), main = "SUR criterion",
xlab = expression(x[1]), ylab = expression(x[2]), color = terrain.colors,
plot.axes = {axis(1); axis(2);
      points(design.grid[,1], design.grid[,2], pch = 21, bg = "white")
      contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid),
      matrix(obj.grid, n.grid), nlevels = 10,
      add=TRUE,drawlabels=TRUE, col = "black")
      contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid),
      matrix(cstineq.grid, n.grid), level = 0, add=TRUE,
      drawlabels=FALSE,lwd=1.5, col = "red")
      contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid),
      matrix(csteq.grid, n.grid), level = 0, add=TRUE,
      drawlabels=FALSE,lwd=1.5, col = "orange")
      }
      )

---

easyEGO  
User-friendly wrapper of the functions fastEGO.nsteps and TREGO.nsteps. Generates initial DOEs and kriging models (objects of class km), and executes nsteps iterations of either EGO or TREGO.
**easyEGO**

**Description**
User-friendly wrapper of the functions `FastEGO.nsteps` and `TREGO.nsteps`. Generates initial DOEs and kriging models (objects of class `km`), and executes `nsteps` iterations of either EGO or TREGO.

**Usage**

```r
easyEGO(
  fun,  # scalar function to be minimized,
  budget,  # total number of calls to the objective and constraint functions,
  lower,  # vector of lower bounds for the variables to be optimized over,
  upper,  # vector of upper bounds for the variables to be optimized over,
  X = NULL,  # initial design of experiments. If not provided, X is taken as a maximin LHD with budget/3 points
  y = NULL,  # initial set of objective observations \( f(X) \). Computed if not provided.
  control = list(trace = 1, seed = 42),  # an optional list of control parameters. See "Details".
  n.cores = 1,  # number of cores for parallel computation
  ...  # additional parameters to be given to `fun`
)
```

**Arguments**

- **fun**: scalar function to be minimized,
- **budget**: total number of calls to the objective and constraint functions,
- **lower**: vector of lower bounds for the variables to be optimized over,
- **upper**: vector of upper bounds for the variables to be optimized over,
- **X**: initial design of experiments. If not provided, X is taken as a maximin LHD with budget/3 points
- **y**: initial set of objective observations \( f(X) \). Computed if not provided.
- **control**: an optional list of control parameters. See "Details".
- **n.cores**: number of cores for parallel computation

**Details**

Does not require knowledge on kriging models (objects of class `km`).

The `control` argument is a list that can supply any of the following components:

- **trace**: between -1 and 3
- **seed**: to fix the seed of the run
- **cov.reestim**: Boolean, if TRUE (default) the covariance parameters are re-estimated at each iteration
- **model.trend**: trend for the GP model
- **lb, ub**: lower and upper bounds for the GP covariance ranges
- **nugget**: optional nugget effect
• covtype: covariance of the GP model (default "matern5_2")
• optim.method: optimisation of the GP hyperparameters (default "BFGS")
• multistart: number of restarts of BFGS
• gpmean.trick, gpmean.freq: Boolean and integer, resp., for the gpmean trick
• scaling: Boolean, activates input scaling
• warping: Boolean, activates output warping
• TR: Boolean, activates TREGO instead of EGO
• trcontrol: list of parameters of the trust region, see TREGO.nsteps
• always.sample: Boolean, activates force observation even if it leads to poor conditioning

Value

A list with components:

• par: the best feasible point
• values: a vector of the objective and constraints at the point given in par,
• history: a list containing all the points visited by the algorithm (X) and their corresponding objectives (y).
• model: the last GP model, class km
• control: full list of control values, see "Details"
• res: the output of either fastEGO.nsteps or TREGO.nsteps

Author(s)

Victor Picheny

References


Examples

```r
library(parallel)
library(DiceOptim)
set.seed(123)

#############################################################
### 10 ITERATIONS OF TREGO ON THE BRANIN FUNCTION, ####
### STARTING FROM A 9-POINTS FACTORIAL DESIGN ####
#############################################################

# a 9-points factorial design, and the corresponding response
ylim=NULL
fun <- branin; d <- 2
budget <- 5*d
lower <- rep(0,d)
```
upper <- rep(1,d)
n.init <- 2*d

control <- list(n.init=2*d, TR=TRUE, nugget=1e-5, trcontrol=list(algo="TREGO"), multistart=1)
res1 <- easyEGO(fun=fun, budget=budget, lower=lower, upper=upper, control=control, n.cores=1)

par(mfrow=c(3,1))
y <- res1$history$y
steps <- res1$res$all.steps
success <- res1$res$all.success
sigma <- res1$res$all.sigma
ymin <- cummin(y)
pch <- rep(1, length(sigma))
col <- rep("red", length(sigma))
pch[which(!steps)] <- 2
col[which(success)] <- "darkgreen"

pch2 <- c(rep(3, n.init), pch)
col2 <- c(rep("black", n.init), col)
plot(y, col=col2, ylim=ylim, pch=pch2, lwd=2, xlim=c(0, budget))
lines(ymin, col="darkgreen")
abline(v=n.init+.5)

plot(n.init + (1:length(sigma)), sigma, xlim=c(0, budget), ylim=c(0, max(sigma)),
pch=pch, col=col, lwd=2, main="TR size")
lines(n.init + (1:length(sigma)), sigma, xlim=c(0, budget))
abline(v=n.init+.5)

plot(NA, xlim=c(0, budget), ylim=c(0, 1), main="x0 (coordinates)"
for (i in 1:d) {
    lines(n.init + (1:nrow(res1$res$all.x0)), res1$res$all.x0[,i])
    points(n.init + (1:nrow(res1$res$all.x0)), res1$res$all.x0[,i], pch=pch, col=col, lwd=2)
}
abline(v=n.init+.5)

par(mfrow=c(1,1))
pairs(res1$model@X, pch=pch2, col=col2)

easyEGO.cst

**EGO algorithm with constraints**

**Description**

User-friendly wrapper of the function **EGO.cst** Generates initial DOEs and kriging models (objects of class *km*), and executes nsteps iterations of EGO methods integrating constraints.
Usage

easyEGO.cst(
  fun,
  constraint,
  n.cst = 1,
  budget,
  lower,
  upper,
  cheapfun = FALSE,
  equality = FALSE,
  X = NULL,
  y = NULL,
  C = NULL,
  control = list(method = "EFI", trace = 1, inneroptim = "genoud", maxit = 100, seed = 42),
  ...
)

Arguments

fun scalar function to be minimized,
constraint vectorial function corresponding to the constraints, see details below,
n.cst number of constraints,
budget total number of calls to the objective and constraint functions,
lower vector of lower bounds for the variables to be optimized over,
upper vector of upper bounds for the variables to be optimized over,
cheapfun optional boolean, TRUE if the objective is a fast-to-evaluate function that does not need a kriging model
equality either FALSE if all constraints are inequalities, else a Boolean vector indicating which are equalities
X initial design of experiments. If not provided, X is taken as a maximin LHD with budget/3 points
y initial set of objective observations \( f(X) \). Computed if not provided.
C initial set of constraint observations \( g(X) \). Computed if not provided.
control an optional list of control parameters. See "Details".
... additional parameters to be given to BOTH the objective \texttt{fun} and constraints.

Details

Does not require knowledge on kriging models (objects of class \texttt{km})

The problem considered is of the form: \( \min f(x) \) s.t. \( g(x) \leq 0, g \) having a vectorial output. By default all its components are supposed to be inequalities, but one can use a Boolean vector in \texttt{equality} to specify which are equality constraints, hence of the type \( g(x) = 0 \). The \texttt{control} argument is a list that can supply any of the following components:
- **method**: choice of constrained improvement function: "AL", "EFI" or "SUR" (see `critEFI`, `critAL`, `critSUR_cst`)

- **trace**: if positive, tracing information on the progress of the optimization is produced.

- **inneroptim**: choice of the inner optimization algorithm: "genoud" or "random" (see `genoud`).

- **maxit**: maximum number of iterations of the inner loop.

- **seed**: to fix the random variable generator

For additional details, see `EGO.cst`.

**Value**

A list with components:

- **par**: the best feasible point

- **values**: a vector of the objective and constraints at the point given in par,

- **history**: a list containing all the points visited by the algorithm (X) and their corresponding objectives (y) and constraints (C)

If no feasible point is found, par returns the most feasible point (in the least square sense).

**Author(s)**

Victor Picheny

Mickael Binois

**References**


J.M. Parr (2012), *Improvement criteria for constraint handling and multiobjective optimization*, University of Southampton.

Examples

#---------------------------------------------------------------------------
# 2D objective function, 3 cases
#---------------------------------------------------------------------------

set.seed(25468)
library(DiceDesign)

n_var <- 2
fun <- goldsteinprice
fun1.cst <- function(x){return(-branin(x) + 25)}

# Constraint function with vectorial output
constraint <- function(x){return(c(fun1.cst(x), fun2.cst(x)))}

# For illustration purposes
n.grid <- 31
test.grid <- expand.grid(X1 = seq(0, 1, length.out = n.grid), X2 = seq(0, 1, length.out = n.grid))
obj.grid <- apply(test.grid, 1, fun)
cst1.grid <- apply(test.grid, 1, fun1.cst)
cst2.grid <- apply(test.grid, 1, fun2.cst)

lower <- rep(0, n_var)
upper <- rep(1, n_var)

#---------------------------------------------------------------------------
# 1- Expected Feasible Improvement criterion, expensive objective function,
# two inequality constraints, 15 observations budget, using genoud
#---------------------------------------------------------------------------

res <- easyEGO.cst(fun=fun, constraint=constraint, n.cst=2, lower=lower, upper=upper, budget=15,
                   control=list(method="EFI", inneroptim="genoud", maxit=20))

cat("best design found:", res$par, "\n")
cat("corresponding objective and constraints:", res$value, "\n")

# Objective function in colour, constraint boundaries in red
# Initial DoE: white circles, added points: blue crosses, best solution: red cross

filled.contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), nlevels = 50,
               matrix(obj.grid, n.grid), main = "Two inequality constraints",
               xlab = expression(x[1]), ylab = expression(x[2]), color = terrain.colors,
               plot.axes = (axis(1); axis(2);
               contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid),
               matrix(cst1.grid, n.grid), level = 0, add=TRUE,
               dl.wd=FALSE, lwd=1.5, col = "red")
               contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid),
               matrix(cst2.grid, n.grid), level = 0, add=TRUE,drawlabels=FALSE,
               lwd=1.5, col = "red")
               points(res$history$x, col = "blue", pch = 4, lwd = 2)
               points(res$par[1], res$par[2], col = "red", pch = 4, lwd = 2, cex=2)"
# 2- Augmented Lagrangian Improvement criterion, expensive objective function,
# one inequality and one equality constraint, 25 observations budget, using random search
#-------------------------------------------------------------------------------------------
res2 <- easyEGO.cst(fun=fun, constraint=constraint, n.cst=2, lower=lower, upper=upper, budget=25,
equality = c(TRUE, FALSE),
control=list(method="AL", inneroptim="random", maxit=100))
cat("best design found:", res2$par, "\n")
cat("corresponding objective and constraints:", res2$value, "\n")

# Objective function in colour, inequality constraint boundary in red, equality
# constraint in orange
# Initial DoE: white circles, added points: blue crosses, best solution: red cross
filled.contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), nlevels = 50,
matrix(obj.grid, n.grid), xlab = expression(x[1]), ylab = expression(x[2]),
main = "Inequality (red) and equality (orange) constraints", color = terrain.colors,
plot.axes = {axis(1); axis(2); contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid),
matrix(cst1.grid, n.grid), level = 0, add=TRUE,
drawlabels=FALSE,lwd=1.5, col = "orange")
contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid),
matrix(cst2.grid, n.grid), level = 0, add=TRUE,
drawlabels=FALSE,lwd=1.5, col = "red")
points(res2$history$X, col = "blue", pch = 4, lwd = 2)
points(res2$par[1], res2$par[2], col = "red", pch = 4, lwd = 2, cex=2)}/

# 3- Stepwise Uncertainty Reduction criterion, fast objective function,
# single inequality constraint, with initial DOE given + 10 observations,
# using genoud
#-------------------------------------------------------------------------------------------
n.init <- 12
design.grid <- round(maximinESE_LHS(lhsDesign(n.init, n_var, seed = 42)$design)$design, 1)
cst2.init <- apply(design.grid, 1, fun2.cst)
res3 <- easyEGO.cst(fun=fun, constraint=fun2.cst, n.cst=1, lower=lower, upper=upper, budget=10,
X=design.grid, C=cst2.init,
cheapfun=TRUE, control=list(method="SUR", inneroptim="genoud", maxit=20))
cat("best design found:", res3$par, "\n")
cat("corresponding objective and constraint:", res3$value, "\n")

# Objective function in colour, inequality constraint boundary in red
# Initial DoE: white circles, added points: blue crosses, best solution: red cross
filled.contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), nlevels = 50,
matrix(obj.grid, n.grid), main = "Single constraint, fast objective",}
EGO.cst

Sequential constrained Expected Improvement maximization and model re-estimation, with a number of iterations fixed in advance by the user

Description

Executes nsteps iterations of EGO methods integrating constraints, based on objects of class km. At each step, kriging models are re-estimated (including covariance parameters re-estimation) based on the initial design points plus the points visited during all previous iterations; then a new point is obtained by maximizing one of the constrained Expected Improvement criteria available.

Usage

EGO.cst(
  model.fun = NULL,
  fun,
  cheapfun = NULL,
  model.constraint,
  constraint,
  equality = FALSE,
  crit = "EFI",
  nsteps,
  lower,
  upper,
  type = "UK",
  cov.reestim = TRUE,
  critcontrol = NULL,
  optimcontrol = list(method = "genoud", threshold = 1e-05, distance = "euclidean",
                      notrace = FALSE),
...)


Arguments

- `model.fun`: object of class `km` corresponding to the objective function,
- `fun`: scalar function to be minimized, corresponding to `model.fun` found by a call to `match.fun`,
- `cheapfun`: optional scalar function to use if the objective is a fast-to-evaluate function (handled next with class `fastfun`, through the use of `match.fun`), which does not need a kriging model, see details below,
- `model.constraint`: either one or a list of models of class `km`, one per constraint,
- `constraint`: vectorial function corresponding to the constraints, see details below,
- `equality`: either `FALSE` if all constraints are for inequalities, else a vector of boolean indicating which are equalities
- `crit`: choice of constrained improvement function: "AL", "EFI" or "SUR", see details below,
- `nsteps`: an integer representing the desired number of iterations,
- `lower`: vector of lower bounds for the variables to be optimized over,
- `upper`: vector of upper bounds for the variables to be optimized over,
- `type`: "SK" or "UK" (by default), depending whether uncertainty related to trend estimation has to be taken into account,
- `cov.reestim`: optional boolean specifying if the kriging hyperparameters should be re-estimated at each iteration,
- `critcontrol`: optional list of parameters for criterion `crit`, see details,
- `optimcontrol`: an optional list of control parameters for optimization of the selected infill criterion:
  - method can be set to "discrete" or "genoud". For "discrete", a matrix `candidate.points` must be given. For "genoud", specific parameters to the chosen method can also be specified (see `genoud`).
  - Options for the `checkPredict` function: threshold (1e-4) and distance (covdist) are used to avoid numerical issues occurring when adding points too close to the existing ones.
  - `notrace` can be set to TRUE to suppress printing of the optimization progresses.
- ... additional parameters to be given to the objective `fun` and constraint.

Details

Extension of the function `EGO.nsteps` to constrained optimization.

The problem considered is of the form: \( \min f(x) \text{ s.t. } g(x) \leq 0, \) \( g \) having a vectorial output. By default all its components are supposed to be inequalities, but one can use a boolean vector in `equality` to specify which are equality constraints. In this case one can modify the tolerance on the constraints using the `tolConstraints` component of `critcontrol`: an optional vector giving a tolerance for each of the constraints (equality or inequality). It is highly recommended to use it
when there are equality constraints since the default tolerance of 0.05 in such case might not be suited.

Available infill criteria with crit are:

- Expected Probability of Feasibily (EFI) crit_EFI,
- Augmented Lagrangian (AL) crit_AL,
- Stepwise Uncertainty Reduction of the excursion volume (SUR) crit_SUR_cst.

Depending on the selected criterion, various parameters are available. More precisions are given in the corresponding help pages.

It is possible to consider a cheap to evaluate objective function submitted to expensive constraints. In this case, provide only a function in cheapfun, with both model.fun and fun to NULL, see examples below.

**Value**

A list with components:

- `par`: a matrix representing the additional points visited during the algorithm,
- `values`: a vector representing the response (objective) values at the points given in par,
- `constraint`: a matrix representing the constraints values at the points given in par,
- `feasibility`: a boolean vector saying if points given in par respect the constraints,
- `nsteps`: an integer representing the desired number of iterations (given in argument),
- `lastmodel.fun`: an object of class `km` corresponding to the objective function,
- `lastmodel.constraint`: one or a list of objects of class `km` corresponding to the last kriging models fitted to the constraints.

If a problem occurs during either model updates or criterion maximization, the last working model and corresponding values are returned.

**Author(s)**

Victor Picheny
Mickael Binois

**References**


J.M. Parr (2012), Improvement criteria for constraint handling and multiobjective optimization, University of Southampton.


See Also
critcst_optimizer, crit_EFI, crit_AL, crit_SUR_cst, easyEGO.cst

Examples

#-----------------------------------------------------------------------------------
# 2D objective function, 3 cases
#-----------------------------------------------------------------------------------

set.seed(25468)
library(DiceDesign)

n_var <- 2
fun <- goldsteinprice
fun1.cst <- function(x){return(-branin(x) + 25)}

# Constraint function with vectorial output
cstfun <- function(x){
  return(c(fun1.cst(x), fun2.cst(x))
}

# For illustration purposes
n.grid <- 31
test.grid <- expand.grid(X1 = seq(0, 1, length.out = n.grid), X2 = seq(0, 1, length.out = n.grid))
obj.grid <- apply(test.grid, 1, fun)
cst1.grid <- apply(test.grid, 1, fun1.cst)
cst2.grid <- apply(test.grid, 1, fun2.cst)

# Initial set of observations and models
n.init <- 12
design.grid <- round(maximinESE_LHS(lhsDesign(n.init, n_var, seed = 42)$design)$design, 1)
obj.init <- apply(design.grid, 1, fun)
cst1.init <- apply(design.grid, 1, fun1.cst)
cst2.init <- apply(design.grid, 1, fun2.cst)
model.fun <- km(~., design = design.grid, response = obj.init)
model.constraint1 <- km(~., design = design.grid, response = cst1.init, lower=c(.2,.2))
model.constraint2 <- km(~., design = design.grid, response = cst2.init, lower=c(.2,.2))
model.constraint <- list(model.constraint1, model.constraint2)
lower <- rep(0, n_var)
upper <- rep(1, n_var)

# 1- Expected Feasible Improvement criterion, expensive objective function, 
# two inequality constraints, 5 iterations, using genoud
#-----------------------------------------------------------------------------------
cstEGO <- EGO.cst(model.fun = model.fun, fun = fun, model.constraint = model.constraint, 
crit = "EFI", constraint = cstfun, equality = FALSE, lower = lower, 
upper = upper, nsteps = 5, optimcontrol = list(method = "genoud", maxit = 20))

# Plots: objective function in colour, constraint boundaries in red 
# Initial DoE: white circles, added points: blue crosses, best solution: red cross
filled.contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), nlevels = 50, 
matrix(obj.grid, n.grid), main = "Two inequality constraints", 
xlab = expression(x[1]), ylab = expression(x[2]), color = terrain.colors,
plot.axes = {axis(1); axis(2);
points(design.grid[,1], design.grid[,2], pch = 21, bg = "white")
contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), 
matrix(cst1.grid, n.grid), level = 0, add=TRUE,drawlabels=FALSE, 
lwd=1.5, col = "red")
contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), 
matrix(cst2.grid, n.grid), level = 0, add=TRUE,drawlabels=FALSE, 
lwd=1.5, col = "red")
points(cstEGO$par, col = "blue", pch = 4, lwd = 2)
}

# 2- Augmented Lagrangian Improvement criterion, expensive objective function, 
# one inequality and one equality constraint, using a discrete set of candidates (grid)
#-----------------------------------------------------------------------------------
cstEGO2 <- EGO.cst(model.fun = model.fun, fun = fun, model.constraint = model.constraint, 
crit = "AL", constraint = cstfun, equality = c(TRUE, FALSE), lower = lower, 
upper = upper, nsteps = 10,
critcontrol = list(tolConstraints = c(2, 0), always.update=TRUE),
optimcontrol=list(method="discrete", candidate.points=as.matrix(test.grid)))

# Plots: objective function in colour, inequality constraint boundary in red, 
# equality constraint in orange 
# Initial DoE: white circles, added points: blue crosses, best solution: red cross
filled.contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), nlevels = 50, 
matrix(obj.grid, n.grid), main = "Inequality (red) and equality (orange) constraints", 
xlab = expression(x[1]), ylab = expression(x[2]), color = terrain.colors,
plot.axes = {axis(1); axis(2);
points(design.grid[,1], design.grid[,2], pch = 21, bg = "white")
contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), 
matrix(cst1.grid, n.grid), level = 0, add=TRUE,drawlabels=FALSE, 
lwd=1.5, col = "orange")
contour(seq(0, 1, length.out = n.grid), seq(0, 1, length.out = n.grid), 
matrix(cst2.grid, n.grid), level = 0, add=TRUE,drawlabels=FALSE, 
lwd=1.5, col = "red")
points(cstEGO2$par, col = "blue", pch = 4, lwd = 2)
EGO.nsteps

Sequential EI maximization and model re-estimation, with a number of iterations fixed in advance by the user

Description

Executes \( n \) steps iterations of the EGO method to an object of class \( km \). At each step, a kriging model is re-estimated (including covariance parameters re-estimation) based on the initial design points plus the points visited during all previous iterations; then a new point is obtained by maximizing the Expected Improvement criterion (EI).

Usage

\[
\text{EGO.nsteps}(\ldots)
\]
model, fun, nsteps, lower, upper, parinit = NULL, control = NULL, kmcontrol = NULL }

**Arguments**

- **model**: an object of class `km`.
- **fun**: the objective function to be minimized.
- **nsteps**: an integer representing the desired number of iterations.
- **lower**: vector of lower bounds for the variables to be optimized over.
- **upper**: vector of upper bounds for the variables to be optimized over.
- **parinit**: optional vector of initial values for the variables to be optimized over.
- **control**: an optional list of control parameters for optimization. One can control "pop.size" (default: \([4+3^\log(\text{nb of variables})]\)), "max.generations" (default: :5), "wait.generations" (default: :2), "BFGSburnin" (default: :0), of the function `genoud`.
- **kmcontrol**: an optional list representing the control variables for the re-estimation of the kriging model. The items are the same as in `km`: penalty, optim.method, parinit, control. The default values are those contained in model, typically corresponding to the variables used in `km` to estimate a kriging model from the initial design points.

**Value**

A list with components:

- **par**: a data frame representing the additional points visited during the algorithm.
- **value**: a data frame representing the response values at the points given in par.
- **npoints**: an integer representing the number of parallel computations (=1 here).
- **nsteps**: an integer representing the desired number of iterations (given in argument).
- **lastmodel**: an object of class `km` corresponding to the last kriging model fitted.

**Note**

Most EGO-like methods (EI algorithms) usually work with Ordinary Kriging (constant trend), by maximization of the expected improvement. Here, the EI maximization is also possible with any linear trend. However, note that the optimization may perform much faster and better when the trend is a constant since it is the only case where the analytical gradient is available.

For more details on kmcontrol, see the documentation of km.
EGO.nsteps

Author(s)
David Ginsbourger
Olivier Roustant

References

See Also
EI, max_EI, EI.grad

Examples

```r
set.seed(123)
###############################################################
### 10 ITERATIONS OF EGO ON THE BRANIN FUNCTION, ####
### STARTING FROM A 9-POINTS FACTORIAL DESIGN ####
###############################################################
# a 9-points factorial design, and the corresponding response
d <- 2
n <- 9
design.fact <- expand.grid(seq(0,1,length=3), seq(0,1,length=3))
names(design.fact)<-c("x1", "x2")
design.fact <- data.frame(design.fact)
names(design.fact)<-c("x1", "x2")
response.branin <- apply(design.fact, 1, branin)
response.branin <- data.frame(response.branin)
names(response.branin) <- "y"
# model identification
fitted.model1 <- km(~1, design=design.fact, response=response.branin, covtype="gauss", control=list(pop.size=50, trace=FALSE), parinit=c(0.5, 0.5))
# EGO n steps
library(rgenoud)
nsteps <- 5 # Was 10, reduced to 5 for speeding up compilation
lower <- rep(0,d)
upper <- rep(1,d)
oEGO <- EGO.nsteps(model=fitted.model1, fun=branin, nsteps=nsteps, lower=lower, upper=upper, control=list(pop.size=20, BFGSburnin=2))
```
```r
print(oEGO$par)
print(oEGO$value)

# graphics
n.grid <- 15 # Was 20, reduced to 15 for speeding up compilation
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
response.grid <- apply(design.grid, 1, branin)
z.grid <- matrix(response.grid, n.grid, n.grid)
contour(x.grid, y.grid, z.grid, 40)
title("Branin function")
points(design.fact[,1], design.fact[,2], pch=17, col="blue")
points(oEGO$par, pch=19, col="red")
text(oEGO$par[,1], oEGO$par[,2], labels=1:nsteps, pos=3)
```

```r
###############################################################
### 20 ITERATIONS OF EGO ON THE GOLDSTEIN-PRICE, ####
### STARTING FROM A 9-POINTS FACTORIAL DESIGN ####
###############################################################
```
EI

Analytical expression of the Expected Improvement criterion

Description

Computes the Expected Improvement at current location. The current minimum of the observations can be replaced by an arbitrary value (plugin), which is useful in particular in noisy frameworks.
Usage

```
EI(
  x,  # a vector representing the input for which one wishes to calculate EI,
  model,  # an object of class km,
  plugin = NULL,  # optional scalar: if provided, it replaces the minimum of the current observations,
  type = "UK",  # "SK" or "UK" (by default), depending whether uncertainty related to trend estimation has to be taken into account,
  minimization = TRUE,  # logical specifying if EI is used in minimization or in maximization,
  envir = NULL,  # an optional environment specifying where to assign intermediate values for future gradient calculations. Default is NULL.
  proxy = FALSE  # an optional Boolean, if TRUE EI is replaced by the kriging mean (to minimize)
)
```

Arguments

- **x**: a vector representing the input for which one wishes to calculate EI,
- **model**: an object of class `km`,
- **plugin**: optional scalar: if provided, it replaces the minimum of the current observations,
- **type**: "SK" or "UK" (by default), depending whether uncertainty related to trend estimation has to be taken into account,
- **minimization**: logical specifying if EI is used in minimization or in maximization,
- **envir**: an optional environment specifying where to assign intermediate values for future gradient calculations. Default is `NULL`.
- **proxy**: an optional Boolean, if `TRUE` EI is replaced by the kriging mean (to minimize)

Value

The expected improvement, defined as

\[ EI(x) := E[(\min(Y(X)) - Y(x))^+ | Y(X) = y(X)], \]

where \( X \) is the current design of experiments and \( Y \) is the random process assumed to have generated the objective function \( y \). If a plugin is specified, it replaces \( \min(Y(X)) \)

in the previous formula.

Author(s)

David Ginsbourger
Olivier Roustant
Victor Picheny

References


See Also

max_EI, EGO.nsteps, qEI

Examples

```r
set.seed(123)
##########################################################################
### EI SURFACE ASSOCIATED WITH AN ORDINARY KRBING MODEL ####
### OF THE BRANIN FUNCTION KNOWN AT A 9-POINTS FACTORIAL DESIGN ####
##########################################################################
# a 9-points factorial design, and the corresponding response
d <- 2; n <- 9
design.fact <- expand.grid(seq(0,1,length=3), seq(0,1,length=3))
names(design.fact)<-c("x1", "x2")
design.fact <- data.frame(design.fact)
names(design.fact)<-c("x1", "x2")
response.branin <- apply(design.fact, 1, branin)
response.branin <- data.frame(response.branin)
names(response.branin) <- "y"
# model identification
fitted.model1 <- km(~1, design=design.fact, response=response.branin,
covtype="gauss", control=list(pop.size=50,trace=FALSE), parinit=c(0.5, 0.5))
# graphics
n.grid <- 12
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
EI.grid <- apply(design.grid, 1, EI, fitted.model1)
z.grid <- matrix(EI.grid, n.grid, n.grid)
contour(x.grid,y.grid,z.grid,25)
title("Expected Improvement for the Branin function known at 9 points")
points(design.fact[,1], design.fact[,2], pch=17, col="blue")
```

EI.grad

*Analytical gradient of the Expected Improvement criterion*
Description

Computes the gradient of the Expected Improvement at the current location. The current minimum of the observations can be replaced by an arbitrary value (plugin), which is useful in particular in noisy frameworks.

Usage

\[
\text{EI.grad}(\ x, \ \\
\text{model}, \ \\
\text{plugin} = \text{NULL}, \ \\
\text{type} = \text{"UK"}, \ \\
\text{minimization} = \text{TRUE}, \ \\
\text{envir} = \text{NULL}, \ \\
\text{proxy} = \text{FALSE})
\]

Arguments

- **x**: a vector representing the input for which one wishes to calculate \( \text{EI} \).
- **model**: an object of class \text{km}.
- **plugin**: optional scalar: if provided, it replaces the minimum of the current observations,
- **type**: Kriging type: “SK” or “UK”
- **minimization**: logical specifying if \text{EI} is used in minimization or in maximization,
- **envir**: an optional environment specifying where to get intermediate values calculated in \text{EI}.
- **proxy**: an optional Boolean, if TRUE \text{EI} is replaced by the kriging mean (to minimize)

Value

The gradient of the expected improvement criterion with respect to \( x \). Returns 0 at design points (where the gradient does not exist).

Author(s)

David Ginsbourger
Olivier Roustant
Victor Picheny

References


See Also

EI

Examples

```r
set.seed(123)
# a 9-points factorial design, and the corresponding response
d <- 2; n <- 9
design.fact <- expand.grid(seq(0,1,length=3), seq(0,1,length=3))
names(design.fact)<-c("x1", "x2")
design.fact <- data.frame(design.fact)
names(design.fact)<-c("x1", "x2")
response.branin <- apply(design.fact, 1, branin)
response.branin <- data.frame(response.branin)
names(response.branin) <- "y"

# model identification
fitted.model1 <- km(~1, design=design.fact, response=response.branin,
covtype="gauss", control=list(pop.size=50,trace=FALSE), parinit=c(0.5, 0.5))

# graphics
n.grid <- 9 # Increase to 50 for a nicer picture
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
#response.grid <- apply(design.grid, 1, branin)
EI.grid <- apply(design.grid, 1, EI,fitted.model1)
#EI.grid <- apply(design.grid, 1, EI.plot,fitted.model1, gr=TRUE)

z.grid <- matrix(EI.grid, n.grid, n.grid)
contour(x.grid,y.grid,z.grid,20)
title("Expected Improvement for the Branin function known at 9 points")
points(design.fact[,1], design.fact[,2], pch=17, col="blue")

# graphics
n.gridx <- 5 # increase to 15 for nicer picture
n.gridy <- 5 # increase to 15 for nicer picture
x.grid2 <- seq(0,1,length=n.gridx)
y.grid2 <- seq(0,1,length=n.gridy)
design.grid2 <- expand.grid(x.grid2, y.grid2)

EI.envir <- new.env()
environment(EI) <- environment(EI.grad) <- EI.envir

for(i in seq(1, nrow(design.grid2) ) )
```

{  
x <- design.grid2[i,]
ei <- EI(x, model=fitted.model1, envir=EI.envir)
eigrad <- EI.grad(x, model=fitted.model1, envir=EI.envir)
if(!is.null(ei))
{
  suppressWarnings(arrows(x$Var1,x$Var2,
    x$Var1 + eigrad[1]*2.2*10e-5, x$Var2 + eigrad[2]*2.2*10e-5,
    length = 0.04, code=2, col="orange", lwd=2))
}
}

---

**EQI**

**Expected Quantile Improvement**

**Description**

Evaluation of the Expected Quantile Improvement (EQI) criterion.

**Usage**

```r
EQI(
  x,
  model,
  new.noise.var = 0,
  beta = 0.9,
  q.min = NULL,
  type = "UK",
  envir = NULL
)
```

**Arguments**

- `x` the input vector at which one wants to evaluate the criterion
- `model` a Kriging model of "km" class
- `new.noise.var` (scalar) noise variance of the future observation. Default value is 0 (noise-free observation).
- `beta` Quantile level (default value is 0.9)
- `q.min` Best kriging quantile. If not provided, this quantity is evaluated.
- `type` Kriging type: "SK" or "UK"
- `envir` environment for saving intermediate calculations and reusing them within EQI.grad

**Value**

Expected Quantile Improvement
Author(s)

Victor Picheny
David Ginsbourger

References


Examples

# EQI SURFACE ASSOCIATED WITH AN ORDINARY KRIGING MODEL
# OF THE BRANIN FUNCTION KNOWN AT A 12-POINT LATIN HYPERCUBE DESIGN

set.seed(421)

# Set test problem parameters
doe.size <- 12
dim <- 2
test.function <- get("branin2")
lower <- rep(0,1,dim)
upper <- rep(1,1,dim)
noise.var <- 0.2

# Generate DOE and response
doe <- as.data.frame(matrix(runif(doe.size*dim),doe.size))
y.tilde <- rep(0, 1, doe.size)
for (i in 1:doe.size) {
y.tilde[i] <- test.function(doe[i,]) + sqrt(noise.var)*rnorm(n=1)
}
y.tilde <- as.numeric(y.tilde)

# Create kriging model
model <- km(y~1, design=doe, response=data.frame(y=y.tilde),
            covtype="gauss", noise.var=rep(noise.var,1,doe.size),
            lower=rep(.1,dim), upper=rep(1,dim), control=list(trace=FALSE))

# Compute actual function and criterion on a grid
n.grid <- 12 # Change to 21 for a nicer picture
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
nt <- nrow(design.grid)
crit.grid <- apply(design.grid, 1, EQI, model=model, new.noise.var=noise.var, beta=.9)
func.grid <- apply(design.grid, 1, test.function)

# Compute kriging mean and variance on a grid
EQI.grad

EQI’s Gradient

Description

Analytical gradient of the Expected Quantile Improvement (EQI) criterion.

Usage

EQI.grad(
x,
model,
new.noise.var = 0,
beta = 0.9,
q.min = NULL,
type = "UK",
...)

names(design.grid) <- c("V1","V2")
pred <- predict(model, newdata=design.grid, type="UK", checkNames = FALSE)
mk.grid <- pred$m
sk.grid <- pred$sd

# Plot actual function
z.grid <- matrix(func.grid, n.grid, n.grid)
filled.contour(x.grid,y.grid, z.grid, nlevels=50, color = rainbow,
plot.axes = {title("Actual function");
points(model@X[,1],model@X[,2],pch=17,col="blue");
axis(1); axis(2))}

# Plot Kriging mean
z.grid <- matrix(mk.grid, n.grid, n.grid)
filled.contour(x.grid,y.grid, z.grid, nlevels=50, color = rainbow,
plot.axes = {title("Kriging mean");
points(model@X[,1],model@X[,2],pch=17,col="blue");
axis(1); axis(2))

# Plot Kriging variance
z.grid <- matrix(sk.grid^2, n.grid, n.grid)
filled.contour(x.grid,y.grid, z.grid, nlevels=50, color = rainbow,
plot.axes = {title("Kriging variance");
points(model@X[,1],model@X[,2],pch=17,col="blue");
axis(1); axis(2))

# Plot EQI criterion
z.grid <- matrix(crit.grid, n.grid, n.grid)
filled.contour(x.grid,y.grid, z.grid, nlevels=50, color = rainbow,
plot.axes = {title("EQI");
points(model@X[,1],model@X[,2],pch=17,col="blue");
axis(1); axis(2))

envir = NULL

Arguments

- **x**: the input vector at which one wants to evaluate the criterion
- **model**: a Kriging model of "km" class
- **new.noise.var**: (scalar) noise variance of the future observation. Default value is 0 (noise-free observation).
- **beta**: Quantile level (default value is 0.9)
- **q.min**: Best kriging quantile. If not provided, this quantity is evaluated.
- **type**: Kriging type: "SK" or "UK"
- **envir**: environment for inheriting intermediate calculations from EQI

Value

Gradient of the Expected Quantile Improvement

Author(s)

Victor Picheny
David Ginsbourger

Examples

```r
set.seed(421)

# Set test problem parameters
doe.size <- 12
dim <- 2
test.function <- get("branin2")
lower <- rep(0,1,dim)
upper <- rep(1,1,dim)
noise.var <- 0.2

# Generate DOE and response
doe <- as.data.frame(matrix(runif(doe.size*dim),doe.size))
y.tilde <- rep(0, 1, doe.size)
for (i in 1:doe.size) {
y.tilde[i] <- test.function(doe[i,]) + sqrt(noise.var)*rnorm(n=1)
}
y.tilde <- as.numeric(y.tilde)

# Create kriging model
model <- km(y~1, design=doe, response=data.frame(y=y.tilde),
            covtype="gauss", noise.var=rep(noise.var,1,doe.size),
            lower=rep(.1,dim), upper=rep(1,dim), control=list(trace=FALSE))
```
# Compute actual function and criterion on a grid
n.grid <- 9  # change to 21 for nicer visuals
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
nt <- nrow(design.grid)
crit.grid <- apply(design.grid, 1, EQI, model=model, new.noise.var=noise.var, beta=.9)
crit.grad <- t(apply(design.grid, 1, EQI.grad, model=model, new.noise.var=noise.var, beta=.9))
z.grid <- matrix(crit.grid, n.grid, n.grid)
contour(x.grid,y.grid, z.grid, 30)
title("EQI and its gradient")
points(model@X[,1],model@X[,2],pch=17,col="blue")
for (i in 1:nt)
{
  x <- design.grid[i,]
suppressWarnings(arrows(x$Var1,x$Var2, x$Var1+crit.grad[i,1]*.2,x$Var2+crit.grad[i,2]*.2,
  length=0.04,code=2,col="orange",lwd=2))
}

---

**fastEGO.nsteps**  
*Sequential EI maximization and model re-estimation, with a number of iterations fixed in advance by the user*

---

**Description**

Executes `nsteps` iterations of the EGO method to an object of class `km`. At each step, a kriging model is re-estimated (including covariance parameters re-estimation) based on the initial design points plus the points visited during all previous iterations; then a new point is obtained by maximizing the Expected Improvement criterion (EI).

**Usage**

```r
fastEGO.nsteps(
  model,
  fun,
  nsteps,
  lower,
  upper,
  control = NULL,
  trace = 0,
  n.cores = 1,
  ...
)
```
Arguments

model  an object of class km,
fun    the objective function to be minimized,
nsteps an integer representing the desired number of iterations,
lower  vector of lower bounds for the variables to be optimized over,
upper  vector of upper bounds for the variables to be optimized over,
control an optional list of control parameters for EGO. One can control
"warping" whether or not a warping is applied to the outputs (default FALSE)
"cov.reestim" whether or not the covariance parameters are estimated at each
step (default TRUE) "gpmean.trick" whether or not EI should be replaced
periodically by the GP mean (default FALSE)
"gpmean.freq" frequency at which EI is replaced by the GP mean (default 1e4)
"always.sample" if TRUE, forces observation even if it creates poor condi-
tioning
trace  between -1 (no trace) and 3 (full messages)
n.cores number of cores used for EI maximisation
...   additional parameters to be given to fun

Value

A list with components:

par    a data frame representing the additional points visited during the algorithm,
value  a data frame representing the response values at the points given in par,
npoints an integer representing the number of parallel computations (=1 here),
nsteps an integer representing the desired number of iterations (given in argument),
lastmodel an object of class km corresponding to the last kriging model fitted. If warping
is true, y values are normalized (warped) and will not match value.

Author(s)

Victor Picheny

References

D.R. Jones, M. Schonlau, and W.J. Welch (1998), Efficient global optimization of expensive black-
T.J. Santner, B.J. Williams, and W.J. Notz (2003), The design and analysis of computer experiments,
Springer.
M. Schonlau (1997), Computer experiments and global optimization, Ph.D. thesis, University of
Waterloo.
See Also

EI, max_crit, EI.grad

Examples

set.seed(123)

# a 9-points factorial design, and the corresponding response
d <- 2
n <- 9
design.fact <- expand.grid(seq(0,1,length=3), seq(0,1,length=3))
names(design.fact)<-c("x1", "x2")
design.fact <- data.frame(design.fact)
names(design.fact)<-c("x1", "x2")
response.branin <- apply(design.fact, 1, branin)
response.branin <- data.frame(response.branin)
names(response.branin) <- "y"

# model identification
fitted.model1 <- km(~1, design=design.fact, response=response.branin,
covtype="gauss", control=list(pop.size=50,trace=FALSE), parinit=c(0.5, 0.5))

# EGO n steps
nsteps <- 5
lower <- rep(0,d)
upper <- rep(1,d)
oEGO <- fastEGO.nsteps(model=fitted.model1, fun=branin, nsteps=nsteps, lower=lower, upper=upper)
print(oEGO$par)
print(oEGO$value)

# graphics
n.grid <- 15 # Was 20, reduced to 15 for speeding up compilation
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
response.grid <- apply(design.grid, 1, branin)
z.grid <- matrix(response.grid, n.grid, n.grid)
contour(x.grid, y.grid, z.grid, 40)
title("Branin function")
points(design.fact[,1], design.fact[,2], pch=17, col="blue")
points(oEGO$par, pch=19, col="red")
text(oEGO$par[,1], oEGO$par[,2], labels=1:nsteps, pos=3)
**Description**

Modification of an R function to be used as with methods predict and update (similar to a `km` object). It creates an S4 object which contains the values corresponding to evaluations of other costly observations. It is useful when an objective can be evaluated fast.

**Usage**

```r
fastfun(fn, design, response = NULL)
```

**Arguments**

- `fn` the evaluator function, found by a call to `match.fun`,
- `design` a data frame representing the design of experiments. The ith row contains the values of the d input variables corresponding to the ith evaluation.
- `response` optional vector (or 1-column matrix or data frame) containing the values of the 1-dimensional output given by the objective function at the design points.

**Value**

An object of class `fastfun-class`.

---

**integration_design_cst**

*Generic function to build integration points (for the SUR criterion)*

**Description**

Modification of the function `integration_design` from the package KrigInv to be usable for SUR-based optimization with constraints.

**Usage**

```r
integration_design_cst(
    integcontrol = NULL,
    lower,
    upper,
    model.fun = NULL,
    model.constraint = NULL,
    equality = FALSE,
    critcontrol = NULL,
    min.prob = 0.001
)
```
Arguments

integcontrol  Optional list specifying the procedure to build the integration points and weights. Many options are possible.
A) If nothing is specified, 100*d points are chosen using the Sobol sequence.
B) One can directly set the field integration.points (p * d matrix) for pre-specified integration points. In this case these integration points and the corresponding vector integration.weights will be used for all the iterations of the algorithm.
C) If the field integration.points is not set then the integration points are renewed at each iteration. In that case one can control the number of integration points n.points (default: 100*d) and a specific distribution distrib. Possible values for distrib are: "sobol", "MC" and "SUR" (default: "sobol").
C.1) The choice "sobol" corresponds to integration points chosen with the Sobol sequence in dimension d (uniform weight).
C.2) The choice "MC" corresponds to points chosen randomly, uniformly on the domain.
C.3) The choice "SUR" corresponds to importance sampling distributions (unequal weights).
   When important sampling procedures are chosen, n.points points are chosen using importance sampling among a discrete set of n.candidates points (default: n.points*10) which are distributed according to a distribution init.distrib (default: "sobol"). Possible values for init.distrib are the space filling distributions "sobol" and "MC" or an user defined distribution "spec". The "sobol" and "MC" choices correspond to quasi random and random points in the domain. If the "spec" value is chosen the user must fill in manually the field init.distrib.spec to specify himself a n.candidates * d matrix of points in dimension d.

lower  Vector containing the lower bounds of the design space.
upper  Vector containing the upper bounds of the design space.
model.fun  object of class km corresponding to the objective functions, or, if the objective function is fast-to-evaluate, a fastfun object,
model.constraint  either one or a list of objects of class km, one for each constraint function,
equality  either FALSE if all constraints are for inequalities, else a vector of boolean indicating which are equalities
critcontrol  optional list of parameters (see crit_SUR_cst); here only the component tolConstraints is used.
min.prob  This argument applies only when importance sampling distributions are chosen.
   For numerical reasons we give a minimum probability for a point to belong to the importance sample. This avoids probabilities equal to zero and importance sampling weights equal to infinity. In an importance sample of M points, the maximum weight becomes 1/min.prob * 1/M.

Value

A list with components:
• integration.points p x d matrix of p points used for the numerical calculation of integrals
• integration.weights a vector of size p corresponding to the weight of each point. If all the points are equally weighted, integration.weights is set to NULL

Author(s)
Victor Picheny
Mickael Binois

References


See Also
crit_SUR_cst KrigInv integration_design

---

kriging.quantile  Kriging quantile

**Description**
Evaluation of a kriging quantile at a new point. To be used in an optimization loop.

**Usage**

```r
kriging.quantile(x, model, beta = 0.1, type = "UK", envir = NULL)
```

**Arguments**

- `x` the input vector at which one wants to evaluate the criterion
- `model` a Kriging model of "km" class
- `beta` Quantile level (default value is 0.1)
- `type` Kriging type: "SK" or "UK"
- `envir` an optional environment specifying where to assign intermediate values for future gradient calculations. Default is NULL.
Value

Kriging quantile

Author(s)

Victor Picheny
David Ginsbourger

Examples

```r
# Set test problem parameters
doe.size <- 12
dim <- 2
test.function <- get("branin2")
lower <- rep(0,1,dim)
upper <- rep(1,1,dim)
noise.var <- 0.2

# Generate DOE and response
doe <- as.data.frame(matrix(runif(doe.size*dim),doe.size))
y.tilde <- rep(0, 1, doe.size)
for (i in 1:doe.size) {
y.tilde[i] <- test.function(doe[i,]) + sqrt(noise.var)*rnorm(n=1)
}
y.tilde <- as.numeric(y.tilde)

# Create kriging model
model <- km(y~1, design=doe, response=data.frame(y=y.tilde),
covtype="gauss", noise.var=rep(noise.var,1,doe.size),
lower=rep(.1,dim), upper=rep(1,dim), control=list(trace=FALSE))

# Compute actual function and criterion on a grid
n.grid <- 12 # Change to 21 for a nicer picture
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
nt <- nrow(design.grid)
crit.grid <- apply(design.grid, 1, kriging.quantile, model=model, beta=.1)
func.grid <- apply(design.grid, 1, test.function)

# Compute kriging mean and variance on a grid
names(design.grid) <- c("V1","V2")
pred <- predict(model, newdata=design.grid, type="UK", checkNames = FALSE)
```
kriging.quantile.grad

Analytical gradient of the Kriging quantile of level beta

Description

Computes the gradient of the Kriging quantile of level beta at the current location. Only available for Universal Kriging with constant trend (Ordinary Kriging).

Usage

kriging.quantile.grad(x, model, beta = 0.1, type = "UK", envir = NULL)

Arguments

x a vector representing the input for which one wishes to calculate kriging.quantile.grad.
model an object of class km.
beta A quantile level (between 0 and 1)
Kriging type: "SK" or "UK"

Value

The gradient of the Kriging mean predictor with respect to x. Returns 0 at design points (where the gradient does not exist).

Author(s)

Victor Picheny
David Ginsbourger

References


See Also

EI.grad

Examples

##########################################################################
### KRIGING QUANTILE SURFACE AND ITS GRADIENT FOR ####
### THE BRANIN FUNCTION KNOWN AT A 12-POINT LATIN HYPERCUBE DESIGN ####
##########################################################################
set.seed(421)

# Set test problem parameters
doe.size <- 12
dim <- 2
test.function <- get("branin2")
lower <- rep(0, 1, dim)
upper <- rep(1, 1, dim)
noise.var <- 0.2

# Generate DOE and response
doe <- as.data.frame(matrix(runif(doe.size*dim), doe.size))
y.tilde <- rep(0, 1, doe.size)
for (i in 1:doe.size) {
  y.tilde[i] <- test.function(doe[i,]) + sqrt(noise.var)*rnorm(n=1)
}
y.tilde <- as.numeric(y.tilde)
max_AEI

Maximizer of the Augmented Expected Improvement criterion function

Description

Maximization, based on the package rgenoud of the Augmented Expected Improvement (AEI) criterion.

Usage

max_AEI(
  model,
  new.noise.var = 0,
  y.min = NULL,
  type = "UK",
  lower,
  upper,
  parinit = NULL,
  control = NULL
)
Arguments

- **model**: a Kriging model of "km" class
- **new.noise.var**: the (scalar) noise variance of the new observation.
- **y.min**: The kriging mean prediction at the current best point (point with smallest kriging quantile). If not provided, this quantity is evaluated inside the AEI function (may increase computational time).
- **type**: Kriging type: "SK" or "UK"
- **lower**: vector containing the lower bounds of the variables to be optimized over
- **upper**: optional vector containing the upper bounds of the variables to be optimized over
- **parinit**: optional vector containing the initial values for the variables to be optimized over
- **control**: optional list of control parameters for optimization. One can control "pop.size" (default: \([N=3*2^\text{dim} \text{ for dim}<6 \text{ and } N=32*\text{dim otherwise}]\), "max.generations" (12), "wait.generations" (2) and "BFGSburnin" (2) of function "genoud" (see genoud). Numbers into brackets are the default values.

Value

A list with components:

- **par**: the best set of parameters found.
- **value**: the value AEI at par.

Author(s)

Victor Picheny
David Ginsbourger

Examples

```r
library(DiceDesign)
set.seed(100)

# Set test problem parameters
doe.size <- 10
dim <- 2
test.function <- get("branin2")
lower <- rep(0,1,dim)
upper <- rep(1,1,dim)
noise.var <- 0.2

# Generate DOE and response
doe <- as.data.frame(IhsDesign(doe.size, dim)$design)
y.tilde <- rep(0, 1, doe.size)
for (i in 1:doe.size) {y.tilde[i] <- test.function(doe[i,])}
```
max_AKG

Maximizer of the Expected Quantile Improvement criterion function

Description

Maximization, based on the package rgenoud of the Expected Quantile Improvement (AKG) criterion.

Usage

max_AKG(
  model,
  new.noise.var = 0,
  type = "UK",
  lower,
  upper,
  parinit = NULL,
)
control = NULL
"

Arguments

model a Kriging model of "km" class
new.noise.var the (scalar) noise variance of an observation. Default value is 0 (noise-free observation).
type Kriging type: "SK" or "UK"
lower vector containing the lower bounds of the variables to be optimized over
upper vector containing the upper bounds of the variables to be optimized over
parinit optional vector containing the initial values for the variables to be optimized over
control optional list of control parameters for optimization. One can control "pop.size" (default: [N=3^dim for dim<6 and N=32*dim otherwise]), "max.generations" (12), "wait.generations" (2) and "BFGSburnin" (2) of function "genoud" (see genoud). Numbers into brackets are the default values

Value

A list with components:

par the best set of parameters found.
value the value AKG at par.

Author(s)

Victor Picheny
David Ginsbourger

Examples

##########################################################################
### AKG SURFACE AND OPTIMIZATION PERFORMED BY GENOUD ####
### FOR AN ORDINARY KRIGING MODEL ####
##########################################################################
set.seed(10)
# Set test problem parameters
doe.size <- 10
dim <- 2
test.function <- get("branin2")
lower <- rep(0,1,dim)
upper <- rep(1,1,dim)
noise.var <- 0.2

# Generate DOE and response
library(DiceDesign)
doe <- as.data.frame(lhsDesign(doe.size, dim)$design)
y tilde <- rep(0, 1, doe.size)
for (i in 1:doe.size) {y tilde[i] <- test.function(doe[i,])
+ sqrt(noise.var)*rnorm(n=1)}
y tilde <- as.numeric(y tilde)

# Create krigeing model
model <- km(y~1, design=doe, response=data.frame(y=y tilde),
covtype="gauss", noise.var=rep(noise.var,1,doe.size),
lower=rep(.1,dim), upper=rep(1,dim), control=list(trace=FALSE))

# Optimisation using max_AKG
res <- max_AKG(model, new.noise.var=noise.var, type = "UK",
lower=c(0,0), upper=c(1,1))
X.genoud <- res$par

## Not run:
# Compute actual function and criterion on a grid
n.grid <- 12 # Change to 21 for a nicer picture
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
names(design.grid) <- c("V1","V2")
n.t <- nrow(design.grid)
crit.grid <- apply(design.grid, 1, AKG, model=model, new.noise.var=noise.var)

# # 2D plots
z.grid <- matrix(crit.grid, n.grid, n.grid)
tit <- "Green: best point found by optimizer"
filled.contour(x.grid,y.grid, z.grid, nlevels=50, color = topo.colors,
plot.axes = {title(tit);points(model@X[,1],model@X[,2],pch=17,col="blue");
points(X.genoud[1],X.genoud[2],pch=17,col="green");
axis(1); axis(2))}

## End(Not run)

max_crit

Maximization of the Expected Improvement criterion

Description

For a number of control$restarts, generates a large number of random samples, then picks the
one with best EI value to start L-BFGS.

Usage

max_crit(
  model,
  type = "UK",
)
Arguments

model an object of class km,
type Kriging type: "SK" or "UK"
lower, upper vectors of lower and upper bounds for the variables to be optimized over,
minimization logical specifying if EI is used in minimization or in maximization,
control optional list of control parameters for optimization. For now only the number of restarts can be set.
proxy Boolean, if TRUE, then EI maximization is replaced by the minimization of the kriging mean.
trcontrol an optional list to activate the Trust-region management (see TREGO.nsteps)
n.cores Number of cores if parallel computation is used

Value

A list with components:

par The best set of parameters found.
value The value of expected improvement at par.

Author(s)

Victor Picheny

Examples

set.seed(123)
library(parallel)

# a 9-points factorial design, and the corresponding response
d <- 2
n <- 9
design.fact <- expand.grid(seq(0,1,length=3), seq(0,1,length=3))
names(design.fact) <- c("x1", "x2")
\begin{verbatim}
design.fact <- data.frame(design.fact)
names(design.fact) <- c("x1", "x2")
response.branin <- apply(design.fact, 1, branin)
response.branin <- data.frame(response.branin)
names(response.branin) <- "y"

# model identification
fitted.model1 <- km(~1, design=design.fact, response=response.branin,
covtype="gauss", control=list(pop.size=50,trace=FALSE), parinit=c(0.5, 0.5))

# EGO one step
lower <- rep(0,d)
upper <- rep(1,d)  # domain for Branin function
oEGO <- max_crit(fitted.model1, lower=lower, upper=upper)
print(oEGO)

# graphics
n.grid <- 20
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
response.grid <- apply(design.grid, 1, branin)
z.grid <- matrix(response.grid, n.grid, n.grid)
contour(x.grid,y.grid,z.grid,40)
title("Branin Function")
points(design.fact[,1], design.fact[,2], pch=17, col="blue")
points(oEGO$par[1], oEGO$par[2], pch=19, col="red")
\end{verbatim}

\begin{itemize}
\item \textbf{max\_EI} \hspace{1cm} \textit{Maximization of the Expected Improvement criterion}
\end{itemize}

\section*{Description}

Given an object of class \texttt{km} and a set of tuning parameters (\texttt{lower}, \texttt{upper}, \texttt{parinit}, and \texttt{control}), \texttt{max\_EI} performs the maximization of the Expected Improvement criterion and delivers the next point to be visited in an EGO-like procedure.

\section*{Usage}

\begin{verbatim}
max\_EI(
    model, 
    plugin = NULL, 
    type = "UK", 
    lower, 
    upper, 
    parinit = NULL, 
    minimization = TRUE, 
    control = NULL
)
\end{verbatim}
Arguments

model an object of class `km`,
plugin optional scalar: if provided, it replaces the minimum of the current observations,
type Kriging type: "SK" or "UK",
lower vector of lower bounds for the variables to be optimized over,
upper vector of upper bounds for the variables to be optimized over,
parinit optional vector of initial values for the variables to be optimized over,
minimization logical specifying if EI is used in minimization or in maximization,
control optional list of control parameters for optimization. One can control "pop.size" (default: \([N=3*2^\text{dim} \text{ for dim}<6 \text{ and } N=32*\text{dim otherwise}]."max.generations" (12), "wait.generations" (2) and "BFGSburnin" (2) of function "genoud" (see `genoud`). Numbers into brackets are the default values.

Details

The latter maximization relies on a genetic algorithm using derivatives, `genoud`. This function plays a central role in the package since it is in constant use in the proposed algorithms. It is important to remark that the information needed about the objective function reduces here to the vector of response values embedded in `model` (no call to the objective function or simulator).

The current minimum of the observations can be replaced by an arbitrary value (plugin), which is usefull in particular in noisy frameworks.

Value

A list with components:

par The best set of parameters found.
value The value of expected improvement at par.

Author(s)

David Ginsbourger
Olivier Roustant
Victor Picheny

References


Examples

```r
set.seed(123)

# "ONE-SHOT" EI-MAXIMIZATION OF THE BRANIN FUNCTION
# KNOWN AT A 9-POINTS FACTORIAL DESIGN

# a 9-points factorial design, and the corresponding response
d <- 2
n <- 9
design.fact <- expand.grid(seq(0,1,length=3), seq(0,1,length=3))
names(design.fact) <- c("x1", "x2")
design.fact <- data.frame(design.fact)
names(design.fact) <- c("x1", "x2")
response.branin <- apply(design.fact, 1, branin)
response.branin <- data.frame(response.branin)
names(response.branin) <- "y"

# model identification
fitted.model1 <- km(~1, design=design.fact, response=response.branin,
covtype="gauss", control=list(pop.size=50,trace=FALSE), parinit=c(0.5, 0.5))

# EGO one step
library(rgenoud)
lower <- rep(0,d)
upper <- rep(1,d)  # domain for Branin function
oEGO <- max_EI(fitted.model1, lower=lower, upper=upper,
control=list(pop.size=20, BFGSburnin=2))
print(oEGO)

# graphics
n.grid <- 20
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
response.grid <- apply(design.grid, 1, branin)
z.grid <- matrix(response.grid, n.grid, n.grid)
contour(x.grid, y.grid, z.grid, 40)
title("Branin Function")
points(design.fact[,1], design.fact[,2], pch=17, col="blue")
points(oEGO$par[1], oEGO$par[2], pch=19, col="red")

# "ONE-SHOT" EI-MAXIMIZATION OF THE CAMELBACK FUNCTION
# KNOWN AT A 16-POINTS FACTORIAL DESIGN

## Not run:
# a 16-points factorial design, and the corresponding response
d <- 2
n <- 16
design.fact <- expand.grid(seq(0,1,length=4), seq(0,1,length=4))
```

max_EI
	names(design.fact)<-c("x1", "x2")
design.fact <- data.frame(design.fact)  
names(design.fact) <- c("x1", "x2")
response.camelback <- apply(design.fact, 1, camelback)
response.camelback <- data.frame(response.camelback)  
names(response.camelback) <- "y"

# model identification
fitted.model1 <- km(~1, design=design.fact, response=response.camelback,  
covtype="gauss", control=list(pop.size=50, trace=FALSE), parinit=c(0.5, 0.5))

# EI maximization
library(rgenoud)
lower <- rep(0,d)
upper <- rep(1,d)
oEGO <- max_EI(fitted.model1, lower=lower, upper=upper,  
control=list(pop.size=20, BFGSburnin=2))
print(oEGO)

# graphics
n.grid <- 20
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)  
response.grid <- apply(design.grid, 1, camelback)
z.grid <- matrix(response.grid, n.grid, n.grid)
contour(x.grid,y.grid,z.grid,40)
title("Camelback Function")
points(design.fact[,1], design.fact[,2], pch=17, col="blue")
points(oEGO$par[1], oEGO$par[2], pch=19, col="red")

## End(Not run)

########################################################################
### "ONE-SHOT" EI-MAXIMIZATION OF THE GOLDSTEIN-PRICE FUNCTION #######
### KNOWN AT A 9-POINTS FACTORIAL DESIGN #######
########################################################################

## Not run:
# a 9-points factorial design, and the corresponding response
n <- 9
d <- 2
design.fact <- expand.grid(seq(0,1,length=3), seq(0,1,length=3))
names(design.fact)<-c("x1", "x2")
design.fact <- data.frame(design.fact)  
names(design.fact) <- c("x1", "x2")
response.goldsteinPrice <- apply(design.fact, 1, goldsteinPrice)
response.goldsteinPrice <- data.frame(response.goldsteinPrice)  
names(response.goldsteinPrice) <- "y"

# model identification
fitted.model1 <- km(~1, design=design.fact, response=response.goldsteinPrice,  
covtype="gauss", control=list(pop.size=50, max.generations=50,  
wait.generations=5, BFGSburnin=10, trace=FALSE), parinit=c(0.5, 0.5), optim.method="gen")
# EI maximization
library(rgenoud)
lower <- rep(0,d); upper <- rep(1,d); # domain for Branin function
oEGO <- max_EQI(fitted.model1, lower=lower, upper=upper, control=list(pop.size=50, max.generations=50, wait.generations=5, BFGSburnin=10))
print(oEGO)

# graphics
n.grid <- 20
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
response.grid <- apply(design.grid, 1, goldsteinPrice)
response.grid <- apply(design.grid, 1, goldsteinPrice)
z.grid <- matrix(response.grid, n.grid, n.grid)
contour(x.grid,y.grid,z.grid,40)
title("Goldstein-Price Function")
points(design.fact[,1], design.fact[,2], pch=17, col="blue")
points(design.fact[,1], design.fact[,2], pch=17, col="blue")
points(oEGO$par[1], oEGO$par[2], pch=19, col="red")

## End(Not run)

max_EQI

Maximizer of the Expected Quantile Improvement criterion function

Description
Maximization, based on the package rgenoud of the Expected Quantile Improvement (EQI) criterion.

Usage

max_EQI(
  model,
  new.noise.var = 0,
  beta = 0.9,
  q.min = NULL,
  type = "UK",
  lower,
  upper,
  parinit = NULL,
  control = NULL
)

Arguments

model a Kriging model of "km" class
new.noise.var the (scalar) noise variance of an observation. Default value is 0 (noise-free observation).
beta  | Quantile level (default value is 0.9)
q.min | The current best kriging quantile. If not provided, this quantity is evaluated inside the EQI function (may increase computational time).
type  | Kriging type: "SK" or "UK"
lower | vector containing the lower bounds of the variables to be optimized over
upper | optional vector containing the upper bounds of the variables to be optimized over
parinit | optional vector containing the initial values for the variables to be optimized over
control | optional list of control parameters for optimization. One can control "pop.size" (default: \[N=3*2^\text{dim} \text{for dim}<6 \text{and N}=32*\text{dim otherwise}\]), "max.generations" (12), "wait.generations" (2) and "BFGSburnin" (2) of function "genoud" (see \texttt{genoud}). Numbers into brackets are the default values

Value

A list with components:

- par | the best set of parameters found.
- value | the value EQI at par.

Author(s)

Victor Picheny
David Ginsbourger

Examples

```r
set.seed(10)

# Set test problem parameters
doe.size <- 10
dim <- 2
test.function <- get("branin2")
lower <- rep(0,1,dim)
upper <- rep(1,1,dim)
noise.var <- 0.2

# Generate DOE and response
doe <- as.data.frame(matrix(runif(doe.size*dim),doe.size))
y.tilde <- rep(0, 1, doe.size)
for (i in 1:doe.size) {y.tilde[i] <- test.function(doe[i,]) + sqrt(noise.var)*rnorm(n=1)}
y.tilde <- as.numeric(y.tilde)

# Create kriging model
model <- km(y~1, design=doe, response=data.frame(y=y.tilde),
            covtype="gauss", noise.var=rep(noise.var,1,doe.size),
```
max_qEI

Maximization of multipoint expected improvement criterion (qEI)

Description

Maximization of the qEI criterion. Two options are available: Constant Liar (CL), and brute force qEI maximization with Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm, or GENetic Optimization Using Derivative (genoud) algorithm.

Usage

max_qEI(
  model,
  npoints,
  lower,
  upper,
  crit = "exact",
  minimization = TRUE,
  optimcontrol = NULL
)

# Optimisation using max_EQI
res <- max_EQI(model, new.noise.var=noise.var, type = "UK",
lower=c(0,0), upper=c(1,1))
X.genoud <- res$par

## Not run:
# Compute actual function and criterion on a grid
n.grid <- 12 # Change to 21 for a nicer picture
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
names(design.grid) <- c("V1","V2")
nt <- nrow(design.grid)
crit.grid <- apply(design.grid, 1, EQI, model=model, new.noise.var=noise.var, beta=.9)

# 2D plots
z.grid <- matrix(crit.grid, n.grid, n.grid)
tit <- "Green: best point found by optimizer"
filled.contour(x.grid,y.grid, z.grid, nlevels=50, color = rainbow,
plot.axes = {title(tit);points(model@X[,1],model@X[,2],pch=17,col="blue");
points(X.genoud[1],X.genoud[2],pch=17,col="green");
axis(1); axis(2))

## End(Not run)
Arguments

- **model**: an object of class `km`.
- **npoints**: an integer representing the desired number of iterations.
- **lower**: vector of lower bounds.
- **upper**: vector of upper bounds.
- **crit**: "exact", "CL" : a string specifying the criterion used. "exact" triggers the maximization of the multipoint expected improvement at each iteration (see `max_qEI`), "CL" applies the Constant Liar heuristic.
- **minimization**: logical specifying if the qEI to be maximized is used in minimization or in maximization.
- **optimcontrol**: an optional list of control parameters for optimization. See details.

Details

- CL is a heuristic method. First, the regular Expected Improvement EI is maximized (max_EI). Then, for the next points, the Expected Improvement is maximized again, but with an artificially updated Kriging model. Since the response values corresponding to the last best point obtained are not available, the idea of CL is to replace them by an arbitrary constant value L (a "lie") set by the user (default is the minimum of all currently available observations).

- The BFGS algorithm is implemented in the standard function `optim`. Analytical formulae of qEI and its gradient `qEI.grad` are used. The nStarts starting points are by default sampled with respect to the regular EI (sampleFromEI) criterion.

- The "genoud" method calls the function `genoud` using analytical formulae of qEI and its gradient `qEI.grad`.

The parameters of list optimcontrol are:

- **optimcontrol$method**: "BFGS" (default), "genoud" ; a string specifying the method used to maximize the criterion (irrelevant when crit is "CL" because this method always uses genoud),
- when crit="CL" :
  + **optimcontrol$parinit**: optional matrix of initial values (must have model@d columns, the number of rows is not constrained),
  + **optimcontrol$L**: "max", "min", "mean" or a scalar value specifying the liar ; "min" takes model@min,"max" takes model@max, "mean" takes the prediction of the model ; When L is NULL, "min" is taken if minimization==TRUE, else it is "max".
  + The parameters of function `genoud`. Main parameters are: "pop.size" (default: [N=3*2^model@d for dim<6 and N=32*model@d otherwise]), "max.generations" (default: 12), "wait.generations" (default: 2) and "BFGSburnin" (default: 2).

- when optimcontrol$method = "BFGS":
  + **optimcontrol$nStarts** (default: 4),
  + **optimcontrol$fastCompute**: if TRUE (default), a fast approximation method based on a semi-analytic formula is used, see [Marmin 2014] for details,
  + **optimcontrol$samplingFun**: a function which sample a batch of starting point (default : sampleFromEI),
max_qEI

+ optimcontrol$parinit: optional 3d-array of initial (or candidate) batches (for all k, parinit[,k] is a matrix of size npoints*model@d representing one batch). The number of initial batches (length(parinit[1,1])) is not constrained and does not have to be equal to nStarts. If there is too few initial batches for nStarts, missing batches are drawn with samplingFun (default : NULL).
- when optimcontrol$method = "genoud":
+ optimcontrol$fastCompute: if TRUE (default), a fast approximation method based on a semi-analytic formula is used, see [Marmin 2014] for details,
+ optimcontrol$parinit: optional matrix of candidate starting points (one row corresponds to one point),
- The parameters of the genoud function. Main parameters are "pop.size" (default: [50*(model@d)*(npoints)]), "max.generations" (default: 5), "wait.generations" (default: 2), "BFGSburnin" (default: 2).

Value

A list with components:

par A matrix containing the npoints input vectors found.
value A value giving the qEI computed in par.

Author(s)

Sebastien Marmin
Clement Chevalier
David Ginsbourger

References

C. Chevalier and D. Ginsbourger (2014) Learning and Intelligent Optimization - 7th International Conference, Lion 7, Catania, Italy, January 7-11, 2013, Revised Selected Papers, chapter Fast computation of the multipoint Expected Improvement with applications in batch selection, pages 59-69, Springer.


See Also

qEI, qEI.grad
Examples

```r
set.seed(000)
# 3-points EI maximization.
# 9-points factorial design, and the corresponding response
d <- 2
n <- 9
design.fact <- expand.grid(seq(0,1,length=3), seq(0,1,length=3))
names(design.fact)<-c("x1", "x2")
design.fact <- data.frame(design.fact)
names(design.fact)<-c("x1", "x2")
response.branin <- apply(design.fact, 1, branin)
response.branin <- data.frame(response.branin)
names(response.branin) <- "y"
lower <- c(0,0)
upper <- c(1,1)

# number of point in the batch
batchSize <- 3

# model identification
fitted.model <- km(~1, design=design.fact, response=response.branin,
covtype="gauss", control=list(pop.size=50,trace=FALSE), parinit=c(0.5, 0.5))

# maximization of qEI
# With a multistarted BFGS algorithm
maxBFGS <- max_qEI(model = fitted.model, npoints = batchSize, lower = lower, upper = upper,
crit = "exact", optimcontrol=list(nStarts=3,method = "BFGS"))

# comparison
print(maxBFGS$value)
## Not run:
# With a genetic algorithm using derivatives
maxGen <- max_qEI(model = fitted.model, npoints = batchSize, lower = lower, upper = upper,
crit = "exact", optimcontrol=list(nStarts=3,method = "genoud",pop.size=100,max.generations = 15))
# With the constant liar heuristic
maxCL <- max_qEI(model = fitted.model, npoints = batchSize, lower = lower, upper = upper,
crit = "CL", optimcontrol=list(pop.size=20))
print(maxGen$value)
print(maxCL$value)
## End(Not run)
```

`min_quantile`  
Minimization of the Kriging quantile.
**Description**

Minimization, based on the package rgenoud of the kriging quantile.

**Usage**

```r
min_quantile(
  model,  
  beta = 0.1,  
  type = "UK",  
  lower,  
  upper,  
  parinit = NULL,  
  control = NULL
)
```

**Arguments**

- `model`: a Kriging model of "km" class
- `beta`: Quantile level (default value is 0.1)
- `type`: Kriging type: "SK" or "UK"
- `lower`: vector containing the lower bounds of the variables to be optimized over
- `upper`: vector containing the upper bounds of the variables to be optimized over
- `parinit`: optional vector containing the initial values for the variables to be optimized over
- `control`: optional list of control parameters for optimization. One can control "pop.size" (default: \[N=3*2^\text{dim for dim<6 and N=32*dim otherwise}\]), "max.generations" (12), "wait.generations" (2) and "BFGSburnin" (2) of function "genoud" (see `genoud`). Numbers into brackets are the default values

**Value**

A list with components:

- `par`: the best set of parameters found.
- `value`: the value of the kriging quantile at par.

**Author(s)**

Victor Picheny
David Ginsbourger

**Examples**

```r
##########################################################################  
### KRIGING QUANTILE SURFACE AND OPTIMIZATION PERFORMED BY GENOUD ####
```
### FOR AN ORDINARY KRIGING MODEL

### OF THE BRANIN FUNCTION KNOWN AT A 12-POINT LATIN HYPERCUBE DESIGN

```r
set.seed(10)

# Set test problem parameters
doe.size <- 10
dim <- 2
test.function <- get("branin2")
lower <- rep(0,1,dim)
upper <- rep(1,1,dim)
noise.var <- 0.2

# Generate DOE and response
doe <- as.data.frame(matrix(runif(doe.size*dim),doe.size))
y.tilde <- rep(0, 1, doe.size)
for (i in 1:doe.size) {y.tilde[i] <- test.function(doe[i,]) + sqrt(noise.var)*rnorm(n=1)}
y.tilde <- as.numeric(y.tilde)

# Create kriging model
model <- km(y~1, design=doe, response=data.frame(y=y.tilde),
            covtype="gauss", noise.var=rep(noise.var,1,doe.size),
            lower=rep(.1,dim), upper=rep(1,dim), control=list(trace=FALSE))

# Optimisation using max_kriging.quantile
res <- min_quantile(model, beta=0.1, type = "UK", lower=c(0,0), upper=c(1,1))
X.genoud <- res$par

# Compute actual function and criterion on a grid
n.grid <- 12 # Change to 21 for a nicer picture
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
names(design.grid) <- c("V1","V2")
nt <- nrow(design.grid)
crit.grid <- apply(design.grid, 1, kriging.quantile, model=model, beta=.1)

# 2D plots
z.grid <- matrix(crit.grid, n.grid, n.grid)
tit <- "Green: best point found by optimizer"
filled.contour(x.grid,y.grid, z.grid, nlevels=50, color = topo.colors,
               plot.axes = {title(tit);points(model@X[,1],model@X[,2],pch=17,col="blue");
                           points(X.genoud[1],X.genoud[2],pch=17,col="green");
                           axis(1); axis(2))}
```

---

**noisy.optimizer**

*Optimization of homogenously noisy functions based on Kriging*
noisy.optimizer

Description

Sequential optimization of kriging-based criterion conditional on noisy observations, with model update after each evaluation. Eight criteria are proposed to choose the next observation: random search, sequential parameter optimization (SPO), reinterpolation, Expected Improvement (EI) with plugin, Expected Quantile Improvement (EQI), quantile minimization, Augmented Expected Improvement (AEI) and Approximate Knowledge Gradient (AKG). The criterion optimization is based on the package rgenoud.

Usage

noisy.optimizer(
  optim.crit, 
  optim.param = NULL, 
  model, 
  n.ite, 
  noise.var = NULL, 
  funnoise, 
  lower, 
  upper, 
  parinit = NULL, 
  control = NULL, 
  CovReEstimate = TRUE, 
  NoiseReEstimate = FALSE, 
  nugget.LB = 1e-05, 
  estim.model = NULL, 
  type = "UK"
)

Arguments

optim.crit String defining the criterion to be optimized at each iteration. Possible values are: "random.search", "SPO", "reinterpolation", "EI.plugin", "EQI", "min.quantile", "AEI", "AKG".

optim.param List of parameters for the chosen criterion. For "EI.plugin": optim.param$plugin.type is a string defining which plugin is to be used. Possible values are "ytilde", "quantile" and "other". If "quantile" is chosen, optim.param$quantile defines the quantile level. If "other" is chosen, optim.param$plugin directly sets the plugin value.

For "EQI": optim.param$quantile defines the quantile level. If not provided, default value is 0.9.

For "min.quantile": optim.param$quantile defines the quantile level. If not provided, default value is 0.1.

For "AEI": optim.param$quantile defines the quantile level to choose the current best point. If not provided, default value is 0.75.

model a Kriging model of "km" class

n.ite Number of iterations
noisy.optimizer

noise.var  Noise variance (scalar). If noiseReEstimate=TRUE, it is an initial guess for the unknown variance (used in optimization).

funnoise  objective (noisy) function

lower  vector containing the lower bounds of the variables to be optimized over

upper  vector containing the upper bounds of the variables to be optimized over

parinit  optional vector of initial values for the variables to be optimized over

control  optional list of control parameters for optimization. One can control "pop.size" (default: \([N=3*2^\text{dim} \text{ for } \text{dim}<6 \text{ and } N=32*\text{dim otherwise}])\), "max.generations" (N), "wait.generations" (2) and "BFGSburnin" (0) of function "genoud" (see genoud). Numbers into brackets are the default values

CovReEstimate  optional boolean specifying if the covariance parameters should be re-estimated at every iteration (default value = TRUE)

NoiseReEstimate  optional boolean specifying if the noise variance should be re-estimated at every iteration (default value = FALSE)

nugget.LB  optional scalar of minimal value for the estimated noise variance. Default value is 1e-5.

estim.model  optional kriging model of "km" class with homogeneous nugget effect (no noise.var). Required if noise variance is reestimated and the initial "model" has heterogeneous noise variances.

type  "SK" or "UK" for Kriging with known or estimated trend

Value

A list with components:

model  the current (last) kriging model of "km" class

best.x  The best design found

best.y  The objective function value at best.x

best.index  The index of best.x in the design of experiments

history.x  The added observations

history.y  The added observation values

history.hyperparam  The history of the kriging parameters

estim.model  If noiseReEstimate=TRUE, the current (last) kriging model of "km" class for estimating the noise variance.

history.noise.var  If noiseReEstimate=TRUE, the history of the noise variance estimate.

Author(s)

Victor Picheny
References

V. Picheny and D. Ginsbourger (2013), Noisy kriging-based optimization methods: A unified implementation within the DiceOptim package, *Computational Statistics & Data Analysis*

Examples

```r
set.seed(10)
library(DiceDesign)
# Set test problem parameters
doe.size <- 9
dim <- 2
test.function <- get("branin2")
lower <- rep(0,1,dim)
upper <- rep(1,1,dim)
noise.var <- 0.1

# Build noisy simulator
funnoise <- function(x)
{  f.new <- test.function(x) + sqrt(noise.var)*rnorm(n=1)
    return(f.new)
}

# Generate DOE and response
doe <- as.data.frame(lhsDesign(doe.size, dim)$design)
y.tilde <- funnoise(doe)

# Create kriging model
model <- km(y~1, design=doe, response=data.frame(y=y.tilde),
                  covtype="gauss", noise.var=rep(noise.var,1,doe.size),
                  lower=rep(.1,dim), upper=rep(1,dim), control=list(trace=FALSE))

# Optimisation with noisy.optimizer (n.ite can be increased)
n.ite <- 2
optim.param <- list()
optim.param$quantile <- .9
optim.result <- noisy.optimizer(optim.crit="EQI", optim.param=optim.param, model=model,
n.ite=n.ite, noise.var=noise.var, funnoise=funnoise, lower=lower, upper=upper,
NoiseReEstimate=FALSE, CovReEstimate=FALSE)

new.model <- optim.result$model
best.x <- optim.result$best.x
new.doe <- optim.result$history.x

## Not run:
##### DRAW RESULTS ######
# Compute actual function on a grid
n.grid <- 12
```
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
names(design.grid) <- c("V1","V2")
nt <- nrow(design.grid)
func.grid <- rep(0,1,nt)
for (i in 1:nt)
{ func.grid[i] <- test.function(x=design.grid[i,])}

# Compute initial and final kriging on a grid
pred <- predict(object=model, newdata=design.grid, type="UK", checkNames = FALSE)
mk.grid1 <- pred$m
sk.grid1 <- pred$sd

pred <- predict(object=new.model, newdata=design.grid, type="UK", checkNames = FALSE)
mk.grid2 <- pred$m
sk.grid2 <- pred$sd

# Plot initial kriging mean
z.grid <- matrix(mk.grid1, n.grid, n.grid)
filled.contour(x.grid,y.grid, z.grid, nlevels=50, color = topo.colors,
plot.axes = {title("Initial kriging mean");
points(model@X[,1],model@X[,2],pch=17,col="black");
axis(1); axis(2))}

# Plot initial kriging variance
z.grid <- matrix(sk.grid1^2, n.grid, n.grid)
filled.contour(x.grid,y.grid, z.grid, nlevels=50, color = topo.colors,
plot.axes = {title("Initial kriging variance");
points(model@X[,1],model@X[,2],pch=17,col="black");
axis(1); axis(2))}

# Plot final kriging mean
z.grid <- matrix(mk.grid2, n.grid, n.grid)
filled.contour(x.grid,y.grid, z.grid, nlevels=50, color = topo.colors,
plot.axes = {title("Final kriging mean");
points(new.model@X[,1],new.model@X[,2],pch=17,col="black");
axis(1); axis(2))}

# Plot final kriging variance
z.grid <- matrix(sk.grid2^2, n.grid, n.grid)
filled.contour(x.grid,y.grid, z.grid, nlevels=50, color = topo.colors,
plot.axes = {title("Final kriging variance");
points(new.model@X[,1],new.model@X[,2],pch=17,col="black");
axis(1); axis(2))}

# Plot actual function and observations
z.grid <- matrix(func.grid, n.grid, n.grid)
tit <- "Actual function - Black: initial points; red: added points"
filled.contour(x.grid,y.grid, z.grid, nlevels=50, color = topo.colors,
plot.axes = {title(tit);points(model@X[,1],model@X[,2],pch=17,col="black");
points(new.doe[1,],new.doe[2,],pch=15,col="red");
axis(1); axis(2))"
## Example 2: 3 Optimization Steps Using EQI with Unknown Noise and Unknown Covariance Parameters for the Branin Function

### Description
Strongly multimodal constraint function from Parr et al. (standardized version)

### Usage
```
ParrConstraint(x)
```

### Arguments
- `x`: a 2-dimensional vector or a two-column matrix specifying the location(s) where the function is to be evaluated.

### Value
A scalar
Examples

```r
n.grid <- 20
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
response.grid <- apply(design.grid, 1, ParrConstraint)
z.grid <- matrix(response.grid, n.grid, n.grid)
contour(x.grid,y.grid,z.grid,40)
title("Parr constraint function")
```

qEGO.nsteps

Sequential multipoint Expected improvement (qEI) maximizations and model re-estimation

Description

Executes `nsteps` iterations of the multipoint EGO method to an object of class `km`. At each step, a kriging model (including covariance parameters) is re-estimated based on the initial design points plus the points visited during all previous iterations; then a new batch of points is obtained by maximizing the multipoint Expected Improvement criterion (qEI).

Usage

```r
qEGO.nsteps(
  fun,
  model,
  npoints,
  nsteps,
  lower = rep(0, model@d),
  upper = rep(1, model@d),
  crit = "exact",
  minimization = TRUE,
  optimcontrol = NULL,
  cov.reestim = TRUE,
  ...
)
```

Arguments

- `fun`: the objective function to be optimized,
- `model`: an object of class `km`,
- `npoints`: an integer representing the desired batchsize,
- `nsteps`: an integer representing the desired number of iterations,
- `lower`: vector of lower bounds for the variables to be optimized over,
- `upper`: vector of upper bounds for the variables to be optimized over,
qEGO.nsteps 89

crit  "exact", "CL" : a string specifying the criterion used. "exact" triggers the maximization of the multipoint expected improvement at each iteration (see max_qEI), "CL" applies the Constant Liar heuristic,

minimization  logical specifying if we want to minimize or maximize fun,

optimcontrol  an optional list of control parameters for the qEI optimization (see details or max_qEI),

cov.reestim  optional boolean specifying if the kriging hyperparameters should be re-estimated at each iteration,

...  optional arguments for fun.

Details

The parameters of list optimcontrol are :

- optimcontrol$method : "BFGS" (default), "genoud" : a string specifying the method used to maximize the criterion (irrelevant when crit is "CL" because this method always uses genoud),
- when crit="CL"
  + optimcontrol$parinit : optional matrix of initial values (must have model@d columns, the number of rows is not constrained),
  + optimcontrol$L : "max", "min", "mean" or a scalar value specifying the liar : "min" takes model@min, "max" takes model@max, "mean" takes the prediction of the model ; When L is NULL, "min" is taken if minimization==TRUE, else it is "max".
  + The parameters of function genoud. Main parameters are : "pop.size" (default : [N=3*2^model@d for dim<6 and N=32*model@d otherwise]), "max.generations" (default : 12), "wait.generations" (default : 2) and "BFGSburnin" (default : 2).
- when optimcontrol$method = "BFGS"
  + optimcontrol$nStarts (default : 4),
  + optimcontrol$fastCompute : if TRUE (default), a fast approximation method based on a semi-analytic formula is used, see [Marmin 2014] for details,
  + optimcontrol$samplingFun : a function which sample a batch of starting point (default : sampleFromEI),
  + optimcontrol$parinit : optional 3d-array of initial (or candidate) batches (for all k, parinit[,k] is a matrix of size npoints*model@d representing one batch). The number of initial batches (length(parinit[1,1,])) is not conrained and does not have to be equal to nStarts. If there is too few initial batches for nStarts, missing batches are drawn with samplingFun (default : NULL),
- when optimcontrol$method = "genoud"
  + optimcontrol$fastCompute : if TRUE (default), a fast approximation method based on a semi-analytic formula is used, see [Marmin 2014] for details,
  + optimcontrol$parinit : optional matrix of candidate starting points (one row corresponds to one point),
  + The parameters of the genoud function. Main parameters are "pop.size" (default : [50*(model@d)*(npoints)]), "max.generations" (default : 5), "wait.generations" (default : 2), "BFGSburnin" (default : 2).
Value

A list with components:

- `par` a data frame representing the additional points visited during the algorithm,
- `value` a data frame representing the response values at the points given in `par`,
- `npoints` an integer representing the number of parallel computations,
- `nsteps` an integer representing the desired number of iterations (given in argument),
- `lastmodel` an object of class `km` corresponding to the last kriging model fitted,
- `history` a vector of size `nsteps` representing the current known optimum at each step.

Author(s)

Sebastien Marmin
Clement Chevalier
David Ginsbourger

References

C. Chevalier and D. Ginsbourger (2014) Learning and Intelligent Optimization - 7th International Conference, Lion 7, Catania, Italy, January 7-11, 2013, Revised Selected Papers, chapter Fast computation of the multipoint Expected Improvement with applications in batch selection, pages 59-69, Springer.


See Also

`qEI`, `max_qEI`, `qEI.grad`
Examples

set.seed(123)

#########################################################################
### 2 ITERATIONS OF EGO ON THE BRANIN FUNCTION, ###
### STARTING FROM A 9-POINTS FACTORIAL DESIGN ###
#########################################################################

# a 9-points factorial design, and the corresponding response
d <- 2
n <- 9
design.fact <- expand.grid(seq(0,1,length=3), seq(0,1,length=3))
names(design.fact)<-c("x1", "x2")
design.fact <- data.frame(design.fact)
names(design.fact)<-c("x1", "x2")
response.branin <- apply(design.fact, 1, branin)
response.branin <- data.frame(response.branin)
names(response.branin) <- "y"

# model identification
fitted.model1 <- km(~1, design=design.fact, response=response.branin,
covtype="gauss", control=list(pop.size=50,trace=FALSE), parinit=c(0.5, 0.5))

# EGO n steps
library(rgenoud)
nsteps <- 2 # increase to 10 for a more meaningful example
lower <- rep(0,d)
upper <- rep(1,d)
npoints <- 3 # The batchsize
oEGO <- qEGO.nsteps(model = fitted.model1, branin, npoints = npoints, nsteps = nsteps,
crit="exact", lower, upper, optimcontrol = NULL)
print(oEGO$par)
print(oEGO$value)
plot(c(1:nsteps),oEGO$history,xlab="Var step",ylab="Current known minimum")

## Not run:
# graphics
n.grid <- 15 # increase to 21 for better picture
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
response.grid <- apply(design.grid, 1, branin)
z.grid <- matrix(response.grid, n.grid, n.grid)
contour(x.grid, y.grid, z.grid, 40)
title("Branin function")
points(design.fact[,1], design.fact[,2], pch=17, col="blue")
points(oEGO$par, pch=19, col="red")
text(oEGO$par[,1], oEGO$par[,2], labels=tcrossprod(rep(1,npoints),1:nsteps)), pos=3)

## End(Not run)
Analytical expression of the multipoint expected improvement (qEI) criterion

**Description**

Computes the multipoint expected improvement criterion.

**Usage**

```r
qEI(
  x,
  model,
  plugin = NULL,
  type = "UK",
  minimization = TRUE,
  fastCompute = TRUE,
  eps = 10^(-5),
  envir = NULL
)
```

**Arguments**

- `x`: a matrix representing the set of input points (one row corresponds to one point) where to evaluate the qEI criterion,
- `model`: an object of class `km`,
- `plugin`: optional scalar: if provided, it replaces the minimum of the current observations,
- `type`: "SK" or "UK" (by default), depending whether uncertainty related to trend estimation has to be taken into account,
- `minimization`: logical specifying if EI is used in minimization or in maximization,
- `fastCompute`: if TRUE, a fast approximation method based on a semi-analytic formula is used (see [Marmin 2014] for details),
- `eps`: the value of epsilon of the fast computation trick. Relevant only if fastComputation is TRUE,
- `envir`: an optional environment specifying where to get intermediate values calculated in qEI.

**Value**

The multipoint Expected Improvement, defined as

\[ qEI(X_{\text{new}}) := E[(\min(Y(X)) - \min(Y(X_{\text{new}})))^+] + Y(X) = y(X)], \]

where \( X \) is the current design of experiments, \( X_{\text{new}} \) is a new candidate design, and \( Y \) is a random process assumed to have generated the objective function \( y \).
Author(s)
Sebastien Marmin
Clement Chevalier
David Ginsbourger

References
C. Chevalier and D. Ginsbourger (2014) Learning and Intelligent Optimization - 7th International Conference, Lion 7, Catania, Italy, January 7-11, 2013, Revised Selected Papers, chapter Fast computation of the multipoint Expected Improvement with applications in batch selection, pages 59-69, Springer.


See Also
EI

Examples

set.seed(007)

# Monte-Carlo validation

# a 4-d, 81-points grid design, and the corresponding response
d <- 4; n <- 3^d
design <- do.call(expand.grid,rep(list(seq(0,1,length=3)),d))
names(design) <- paste("x",1:d,sep="")
y <- data.frame(apply(design, 1, hartman4))
names(y) <- "y"

# learning
model <- km(~1, design=design, response=y, control=list(trace=FALSE))

# pick up 10 points sampled from the 1-point expected improvement
q <- 10
X <- sampleFromEI(model,n=q)
# simulation of the minimum of the kriging random vector at X

# simulation of the minimum of the kriging random vector at X

# simulation of the improvement (minimization)
qImprovement <- (min(model@y)-minYsim)*((min(model@y)-minYsim) > 0)

# empirical expectation of the improvement and confident interval (95%)
eiMC <- mean(qImprovement)
confInterv <- c(eiMC - 1.96*sd(qImprovement)*1/sqrt(nsim),eiMC + 1.96*sd(qImprovement)*1/sqrt(nsim))

# MC estimation of the qEI
print(eiMC)

# qEI with analytical formula
qEI(X,model,fastCompute= FALSE)

# qEI with fast computation trick
qEI(X,model)

t2-t1 # Time of MC computation
t3-t2 # Time of normal computation
t4-t3 # Time of fast computation

---

**qEI.grad**

*Gradient of the multipoint expected improvement (qEI) criterion*

**Description**

Computes an exact or approximate gradient of the multipoint expected improvement criterion

**Usage**

```r
qEI.grad(
  x,
  model,
  plugin = NULL,
  type = "UK",
)```
minimization = TRUE,
fastCompute = TRUE,
eps = 10^(-6),
envir = NULL
)

Arguments

x         a matrix representing the set of input points (one row corresponds to one point)
where to evaluate the gradient,
model     an object of class km,
plugin    optional scalar: if provided, it replaces the minimum of the current observations,
type      "SK" or "UK" (by default), depending whether uncertainty related to trend estimation has to be taken into account,
minimization logical specifying if EI is used in minimization or in maximization,
fastCompute if TRUE, a fast approximation method based on a semi-analytic formula is used
(see [Marmin 2014] for details),
eps       the value of \( \epsilon \) of the fast computation trick. Relevant only if fastComputation is TRUE,
envir     an optional environment specifying where to get intermediate values calculated in qEI.

Value

The gradient of the multipoint expected improvement criterion with respect to \( x \). A 0-matrix is returned if the batch of input points contains twice the same point or a point from the design experiment of the km object (the gradient does not exist in these cases).

Author(s)

Sebastien Marmin
Clement Chevalier
David Ginsbourger

References

C. Chevalier and D. Ginsbourger (2014) Learning and Intelligent Optimization - 7th International Conference, Lion 7, Catania, Italy, January 7-11, 2013, Revised Selected Papers, chapter Fast computation of the multipoint Expected Improvement with applications in batch selection, pages 59-69, Springer.


See Also

qEI

Examples

```r
set.seed(15)
# Example 1 - validation by comparison to finite difference approximations

# a 9-points factorial design, and the corresponding response
d <- 2
n <- 9
design <- expand.grid(seq(0,1,length=3), seq(0,1,length=3))
names(design)<-c("x1", "x2")
design <- data.frame(design)
names(design)<-c("x1", "x2")
y <- apply(design, 1, branin)
y <- data.frame(y)
names(y) <- "y"

# learning
model <- km(~1, design=design, response=y)

# pick up 2 points sampled from the simple expected improvement
q <- 2 # increase to 4 for a more meaningful test
X <- sampleFromEI(model,n=q)

# compute the gradient at the 4-point batch
grad.analytic <- qEI.grad(X,model)
# numerically compute the gradient
grad.numeric <- matrix(NaN,q,d)
eps <- 10^(-6)
EPS <- matrix(0,q,d)
for (i in 1:q) {
  for (j in 1:d) {
    EPS[i,j] <- eps
    grad.numeric[i,j] <- 1/eps*(qEI(X+EPS,model,fastCompute=FALSE)-qEI(X,model,fastCompute=FALSE))
    EPS[i,j] <- 0
  }
}
print(grad.numeric)
print(grad.analytic)

## Not run:
```
sampleFromEI

Sampling points according to the expected improvement criterion

Description

Samples $n$ points from a distribution proportional to the expected improvement (EI) computed from a km object.

Usage

```r
sampleFromEI(
  model, 
  minimization = TRUE, 
  n = 1, 
  initdistrib = NULL, 
  lower = rep(0, model@d), 
  upper = rep(1, model@d), 
  T = NULL 
)
```

Arguments

- `model`: an object of class km.
- `minimization`: logical specifying if EI is used in minimization or in maximization.
- `n`: number of points to be sampled.
- `initdistrib`: matrix of candidate points.
lower  vector of lower bounds,
upper  vector of upper bounds,
T     optional scalar: if provided, it replaces the current minimum (or maximum) of observations.

Value
A n*d matrix containing the sampled points. If NULL, 1000*d points are obtained by latin hypercube sampling.

Author(s)
Sebastien Marmin
Clement Chevalier
David Ginsbourger

References

See Also
EI, km, qEI

Examples

```r
set.seed(004)

# a 9-points factorial design, and the corresponding responses
d <- 2
n <- 9
design.fact <- expand.grid(seq(0,1,length=3), seq(0,1,length=3))
names(design.fact)<-c("x1", "x2")
design.fact <- data.frame(design.fact)
names(design.fact)<-c("x1", "x2")
response.branin <- apply(design.fact, 1, branin)
response.branin <- data.frame(response.branin)
lower <- c(0,0)
upper <- c(1,1)
names(response.branin) <- "y"

# model identification
fitted.model <- km(~1, design=design.fact, response=response.branin,
covtype="gauss", control=list(pop.size=50, trace=FALSE), parinit=c(0.5, 0.5))
```
# sample a 30 point batch
batchSize <- 30
x <- sampleFromEI(model = fitted.model, n = batchSize, lower = lower, upper = upper)

# graphics
# displays the EI criterion, the design points in black and the EI-sampled points in red.
nGrid <- 15
gridAxel <- seq(lower[1],upper[1],length=nGrid)
gridAxe2 <- seq(lower[2],upper[2],length=nGrid)
grid <- expand.grid(gridAxel,gridAxe2)
aa <- apply(grid,1,EI,model=fitted.model)
myMat <- matrix(aa,nrow=nGrid)
image(x = gridAxel, y = gridAxe2, z = myMat,
     col = colorRampPalette(c("darkgray","white"))(5*10),
     ylab = names(design.fact)[1], xlab=names(design.fact)[2],
     main = "Sampling from the expected improvement criterion",
     axes = TRUE, zlim = c(min(myMat), max(myMat)))
contour(x = gridAxel, y = gridAxe2, z = myMat,
        add = TRUE, nlevels = 10)
points(x[,1],x[,2],pch=19,col='red')
points(fitted.model@X[,1],fitted.model@X[,2],pch=19)

---

test_feas_vec  

**Test constraints violation (vectorized)**

description

Test whether a set of constraints are violated or not, depending on their nature (equality or inequality) and tolerance parameters

Usage

test_feas_vec(cst, equality = FALSE, tolConstraints = NULL)

Arguments

cst matrix of constraints (one column for each constraint function)
equality either FALSE or a Boolean vector defining which constraints are treated as equalities
tolConstraints tolerance (vector) for all constraints. If not provided, set to zero for inequalities and 0.05 for equalities

Value

A Boolean vector, TRUE if the point if feasible, FALSE if at least one constraint is violated
Author(s)

Mickael Binois
Victor Picheny

TREGO.nsteps

Trust-region based EGO algorithm.

Description

Executes \textit{nsteps} iterations of the TREGO method to an object of class \texttt{km}. At each step, a kriging model is re-estimated (including covariance parameters re-estimation) based on the initial design points plus the points visited during all previous iterations; then a new point is obtained by maximizing the Expected Improvement criterion (\textit{EI}) over either the entire search space or restricted to a trust region. The trust region is updated at each iteration based on a sufficient decrease condition.

Usage

\begin{verbatim}
TREGO.nsteps(
    model,  
    fun,    
    nsteps, 
    lower,  
    upper,  
    control = NULL,  
    kmcontrol = NULL,  
    trcontrol = NULL,  
    trace = 0,        
    n.cores = 1,      
    ...  
)
\end{verbatim}

Arguments

- \texttt{model}: an object of class \texttt{km},
- \texttt{fun}: the objective function to be minimized,
- \texttt{nsteps}: an integer representing the desired number of iterations,
- \texttt{lower, upper}: vector of lower and upper bounds for the variables to be optimized over,
- \texttt{control}: an optional list of control parameters for optimization. For now only the number of restarts can be set.
- \texttt{kmcontrol}: an optional list representing the control variables for the re-estimation of the kriging model.
TREGO.nsteps

trcontrol an optional list of control parameters for the trust-region scheme: sigma the initial size of the trust region, x0 its initial center, beta the contraction factor, alpha its dilatation factor, kappa the forcing factor, crit the criterion used inside the TR (either "EI" or "gpmean"), GLratio number of consecutive global and local steps, algo either "TREGO" or "TRIKE", minsigma minimal sigma value, maxsigma maximal sigma value, minEI stopping criterion for TRIKE, local.model Boolean; if TRUE, a local model is used within the trust region, local.trend, local.covtype trend and covariance for the local model, n.local.min minimal number of points used to build the local model,

trace between -1 (no trace) and 3 (full messages)

n.cores number of cores used for EI maximisation

... additional parameters to be given to fun

Value

A list with components:

par a data frame representing the additional points visited during the algorithm,
value a data frame representing the response values at the points given in par,
npoints an integer representing the number of parallel computations (=1 here),
nsteps an integer representing the desired number of iterations (given in argument),
lastmodel an object of class km corresponding to the last kriging model fitted. If warping is true, y values are normalized (warped) and will not match value.
all.success a vector of Boolean indicating the successful steps according to the sufficient decrease condition
all.steps a vector of Boolean indicating which steps were global
all.sigma history of trust region size
all.x0 history of trust region centers
local.model if trcontrol$local.model=TRUE, the latest local model

Author(s)

Victor Picheny

References


See Also

EI, max_crit, EI.grad
Examples

```r
set.seed(123)

#############################################################
### 10 ITERATIONS OF TREGO ON THE BRANIN FUNCTION, ####
### STARTING FROM A 9-POINTS FACTORIAL DESIGN ####
#############################################################

# a 9-points factorial design, and the corresponding response
d <- 2
n <- 9
design.fact <- expand.grid(seq(0,1,length=3), seq(0,1,length=3))
names(design.fact)<-c("x1", "x2")
design.fact <- data.frame(design.fact)
names(design.fact)<-c("x1", "x2")
response.branin <- apply(design.fact, 1, branin)
response.branin <- data.frame(response.branin)
names(response.branin) <- "y"

# model identification
fitted.model1 <- km(~1, design=design.fact, response=response.branin,
covtype="gauss", control=list(pop.size=50,trace=FALSE), parinit=c(0.5, 0.5))

# TREGO n steps
nsteps <- 5
lower <- rep(0, d)
upper <- rep(1, d)
oEGO <- TREGO.nsteps(model=fitted.model1, fun=branin, nsteps=nsteps,
lower=lower, upper=upper)
print(oEGO$par)
print(oEGO$value)

# graphics
n.grid <- 15 # Was 20, reduced to 15 for speeding up compilation
x.grid <- y.grid <- seq(0,1,length=n.grid)
design.grid <- expand.grid(x.grid, y.grid)
response.grid <- apply(design.grid, 1, branin)
z.grid <- matrix(response.grid, n.grid, n.grid)
contour(x.grid, y.grid, z.grid, 40)
title("Branin function")
points(design.fact[,1], design.fact[,2], pch=17, col="blue")
points(oEGO$par[,1], oEGO$par[,2], col="red")
text(oEGO$par[,1], oEGO$par[,2], labels=1:nsteps, pos=3)
```

update_km_noisyEGO

Update of one or two Kriging models when adding new observation
**Description**

Update of a noisy Kriging model when adding new observation, with or without covariance parameter re-estimation. When the noise level is unknown, a twin model "estim.model" is also updated.

**Usage**

```r
update_km_noisyEGO(
  model,
  x.new,
  y.new,
  noise.var = 0,
  type = "UK",
  add.obs = TRUE,
  index.in.DOE = NULL,
  CovReEstimate = TRUE,
  NoiseReEstimate = FALSE,
  estim.model = NULL,
  nugget.LB = 1e-05
)
```

**Arguments**

- `model` a Kriging model of "km" class
- `x.new` a matrix containing the new points of experiments
- `y.new` a matrix containing the function values on the points NewX
- `noise.var` scalar: noise variance
- `type` kriging type: "SK" or "UK"
- `add.obs` boolean: if TRUE, the new point does not exist already in the design of experiment `model@X`
- `index.in.DOE` optional integer: if add.obs=TRUE, it specifies the index of the observation in `model@X` corresponding to `x.new`
- `CovReEstimate` optional boolean specifying if the covariance parameters should be re-estimated (default value = TRUE)
- `NoiseReEstimate` optional boolean specifying if the noise variance should be re-estimated (default value = TRUE)
- `estim.model` optional input of "km" class. Required if NoiseReEstimate=TRUE, in order to deal with repetitions.
- `nugget.LB` optional scalar: is used to define a lower bound on the noise variance.

**Value**

A list containing:

- `model` The updated Kriging model
- `estim.model` If NoiseReEstimate=TRUE, the updated `estim.model`
- `noise.var` If NoiseReEstimate=TRUE, the re-estimated noise variance
Author(s)

Victor Picheny

References

V. Picheny and D. Ginsbourger (2013), Noisy kriging-based optimization methods: A unified implementation within the DiceOptim package, *Computational Statistics & Data Analysis*
Index

* models
  AEI, 8
  EI, 47
  EI.grad, 49
  kriging.quantile.grad, 63
  qEI, 92
  qEI.grad, 94

* optimization
  qEI, 92
  sampleFromEI, 97

* optimize
  EGO.nsteps, 43
  EI.grad, 49
  fastEGO.nsteps, 56
  kriging.quantile.grad, 63
  max_crit, 69
  max_EI, 71
  max_qEI, 77
  qEGO.nsteps, 88
  qEI.grad, 94
  TREGO.nsteps, 100

* parallel
  qEI, 92

  AEI, 8  
  AEI.grad, 10  
  AKG, 12  
  AKG.grad, 14

  checkPredict, 16, 18, 22, 25, 28, 39
  crit_AL, 18, 19, 21, 26, 29, 35, 40, 41
  crit_EI, 18, 19, 22, 24, 29, 35, 40, 41
  crit_SUR_cst, 18, 19, 22, 26, 27, 35, 40, 41, 60, 61
  critcst_optimizer, 17, 19, 41
  DiceOptim (DiceOptim-package), 2  
  DiceOptim-package, 2
  easyEGO, 30
  easyEGO.cst, 33, 41
  EGO.cst, 33, 35, 38
  EGO.nsteps, 39, 43, 49
  EI, 22, 26, 29, 43, 45, 47, 50, 51, 56, 58, 93, 98, 100, 101
  EI.grad, 45, 49, 58, 64, 101
  EQI, 52
  EQI.grad, 54
  fastEGO.nsteps, 30–32, 56
  fastfun, 17, 18, 21, 25, 28, 39, 58, 60
  genoud, 17, 18, 35, 39, 44, 66, 68, 72, 76, 78, 79, 81, 84, 89
  integration_design, 59
  integration_design_cst, 59
  km, 16, 17, 21, 25, 28, 30–34, 38–40, 43, 44, 48, 50, 56, 57, 59, 60, 63, 70–72, 78, 88, 90, 95, 97, 98, 100, 101
  kriging.quantile, 61
  kriging.quantile.grad, 63
  match.fun, 39, 59
  max_AEI, 65
  max_AKG, 67
  max_crit, 58, 69, 101
  max_EI, 18, 45, 49, 71, 78
  max_EQI, 75
  max_qEI, 77, 78, 89, 90
  min_quantile, 80
  noisy.optimizer, 82
  optim, 78
  ParrConstraint, 87
  qEGO.nsteps, 88
  qEI, 49, 77–79, 88, 90, 92, 92, 95, 96, 98
  DiceOptim-package, 2
qEI.grad, 78, 79, 90, 94
sampleFromEI, 78, 89, 97
test_feas_vec, 99
TREGO.nsteps, 30-32, 70, 100
update_km_noisyEGO, 102