Package 'NMOF'

June 26, 2018

Type Package

Title Numerical Methods and Optimization in Finance

Version 1.4-3

Date 2018-06-23

Maintainer Enrico Schumann <es@enricoschumann.net>

Depends R (>= 2.14)

Imports grDevices, graphics, parallel, stats, utils

Suggests MASS, RUnit, quadprog

Description Functions, examples and data from the book ``Numerical Methods and Optimization in Finance" by M. Gilli, D. Maringer and E. Schumann (2011), ISBN 978-0123756626. The package provides implementations of several optimisation heuristics, such as Differential Evolution, Genetic Algorithms and Threshold Accepting. There are also functions for the valuation of financial instruments, such as bonds and options, and functions that help with stochastic simulations.

License GPL-3

URL http://nmof.net, http://enricoschumann.net/NMOF

LazyLoad yes

LazyData yes

Classification/JEL C61, C63

NeedsCompilation no

Author Enrico Schumann [aut, cre] (<https://orcid.org/0000-0001-7601-6576>)

Repository CRAN

Date/Publication 2018-06-26 20:32:57 UTC

R topics documented:

NMOF-package . . . 3
bracketing . . . 5
bundData . . . 6
bundFuture . . . 7
callCF . . . 9
callHestoncf . . . 11
callMerton . . . 13
colSubset . . . 15
CPPI . . . 16
DEopt . . . 18
divRatio . . . 21
drawdown . . . 22
EuropeanCall . . . 24
fundData . . . 25
GAopt . . . 26
gridSearch . . . 29
LS.info . . . 31
LSopt . . . 33
MA . . . 37
mc . . . 38
minvar . . . 40
NS . . . 42
NSf . . . 44
optionData . . . 45
pm . . . 46
PSopt . . . 48
putCallParity . . . 51
qTable . . . 53
repairMatrix . . . 55
resampleC . . . 57
restartOpt . . . 58
SA.info . . . 60
SAopt . . . 62
showExample . . . 66
TA.info . . . 67
TAopt . . . 69
testFunctions . . . 74
vanillaBond . . . 76
vanillaOptionEuropean . . . 79
xtContractValue . . . 83
xwGauss . . . 84

Index **87**

NMOF-package *Numerical Methods and Optimization in Finance*

Description

Functions, data and other R code from the book 'Numerical Methods and Optimization in Finance'. Comments/corrections/remarks/suggestions are very welcome (please contact the maintainer directly).

Details

The package contains implementations of several optimisation heuristics: Differential Evolution (DEopt), Genetic Algorithms (GAopt), (Stochastic) Local Search (LSopt), Particle Swarm (PSopt) and Threshold Accepting (TAopt). The term heuristic is meant in the sense of general purpose optimisation method.

Dependencies: The package is completely written in R. A number of packages are *suggested*, but they are not necessary to use the **NMOF** package. More specifically, package **MASS** is needed to run the complete example for PSopt and also in one of the vignettes (PSlms). Package **parallel** is optional for functions bracketing, GAopt, gridSearch and restartOpt, and may become an option for other functions. Package **quadprog** is needed for a vignette (TAportfolio) and some tests. Finally, **RUnit** is needed to run the tests in subdirectory 'unitTests'.

Version numbering: package versions are numbered in the form major-minor-patch. The *patch* level is incremented with any published change, which means that a new version has been pushed to http://enricoschumann.net/R/packages/NMOF/. *Minor* version numbers are incremented when a feature is added or an existing feature is substantially revised. (Such changes will be reported in the NEWS file.) The *major* version number will only be increased if there were a new edition of the book.

The source code of the **NMOF** package is also hosted at https://github.com/enricoschumann/NMOF/.

Optimisation:

There are functions for Differential Evolution (DEopt), Genetic Algorithms (GAopt), (Stochastic) Local Search (LSopt), Particle Swarm (PSopt) and Threshold Accepting (TAopt).

Pricing Financial Instruments:

For options: See vanillaOptionEuropean, vanillaOptionAmerican, putCallParity. For pricing methods that use the characteristic function, see callCF.

For bonds and bond futures: See vanillaBond, bundFuture and xtContractValue.

Simulation:

See resampleC and mc.

Data:

See bundData, fundData and optionData.

Author(s)

Enrico Schumann

Maintainer: Enrico Schumann <es@enricoschumann.net>

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. https://www.elsevier.com/books/numerical-methods-and-optimization-in-finance/gilli/978-0-12-375662-6

Schumann, E. (2017) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

Examples

```
## Not run:
library("NMOF")

## overview
packageDescription("NMOF")
help(package = "NMOF")

## code from the book
showExample("equations.R")
showExample("exampleLS.R", chapter = 13)

## show NEWS file
news(Version >= "0.40-0", package = "NMOF")

## vignettes
vignette(package = "NMOF")
nss <- vignette("DEnss", package = "NMOF")
print(nss)
edit(nss)

## book website
browseURL("http://nmof.net")
browseURL("http://enricoschumann.net/NMOF/")

## more examples
file.show(system.file("NMOFex/README",   package = "NMOF"))
file.show(system.file("NMOFex/NMOFman.R", package = "NMOF"))

## unit tests
file.show(system.file("unitTests/test_results.txt",   package = "NMOF"))

## End(Not run)
test.rep <- readLines(system.file("unitTests/test_results.txt",   package = "NMOF"))
nt <- gsub(".*\\(([0-9]+) checks?\\).*", "\\1",
           test.rep[grep("\\(\\d+ checks?\\)", test.rep)])
message("Number of unit tests: ", sum(as.numeric(nt)))
```

bracketing *Zero-Bracketing*

Description

Bracket the zeros (roots) of a univariate function

Usage

```
bracketing(fun, interval, ...,
           lower = min(interval), upper = max(interval),
           n = 20L,
           method = c("loop", "vectorised", "multicore", "snow"),
           mc.control = list(), cl = NULL)
```

Arguments

`fun`	a univariate function; it will be called as `fun(x, ...)` with `x` being a numeric vector
`interval`	a numeric vector, containing the end-points of the interval to be searched
`...`	further arguments passed to `fun`
`lower`	lower end-point. Ignored if `interval` is specified.
`upper`	upper end-point. Ignored if `interval` is specified.
`n`	the number of function evaluations. Must be at least 2 (in which case `fun` is evaluated only at the end-points); defaults to 20.
`method`	can be `loop` (the default), `vectorised`, `multicore` or `snow`. See Details.
`mc.control`	a list containing settings that will be passed to `mclapply` if `method` is `multicore`. Must be a list of named elements. See the documentation of `mclapply` in package **parallel**.
`cl`	default is `NULL`. If `method` is `snow`, this must be a cluster object or an integer (the number of cores to be used). See the documentation of packages **parallel** and **snow**.

Details

`bracketing` evaluates `fun` at equal-spaced values of `x` between (and including) `lower` and `upper`. If the sign of `fun` changes between two consecutive `x`-values, `bracketing` reports these two `x`-values as containing ('bracketing') a root. There is no guarantee that there is only one root within a reported interval. `bracketing` will not narrow the chosen intervals.

The argument `method` determines how `fun` is evaluated. Default is `loop`. If `method` is `"vectorised"`, `fun` must be written such that it can be evaluated for a vector `x` (see Examples). If `method` is `multicore`, function `mclapply` from package **parallel** is used. Further settings for `mclapply` can be passed through the list `mc.control`. If `multicore` is chosen but the functionality is not available (eg, currently on Windows), then `method` will be set to `loop` and a warning is issued. If `method` is

snow, function clusterApply from package **parallel** is used. In this case, the argument cl must either be a cluster object (see the documentation of clusterApply) or an integer. If an integer, a cluster will be set up via makeCluster(c(rep("localhost", cl)), type = "SOCK"), and stopCluster is called when the function is exited. If snow is chosen but the package is not available or cl is not specified, then method will be set to loop and a warning is issued. In case that cl is a cluster object, stopCluster will not be called automatically.

Value

A numeric matrix with two columns, named *lower* and *upper*. Each row contains one interval that contains at least one root. If no roots were found, the matrix has zero rows.

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

See Also

uniroot (in package **stats**)

Examples

```
## Gilli/Maringer/Schumann (2011), p. 290
testFun <- function(x)
    cos(1/x^2)

bracketing(testFun, interval = c(0.3, 0.9), n = 26L)
bracketing(testFun, interval = c(0.3, 0.9), n = 26L, method = "vectorised")
```

---

bundData *German Government Bond Data*

Description

A sample of data on 44 German government bonds. Contains ISIN, coupon, maturity and dirty price as of 2010-05-31.

Usage

```
bundData
```

Format

bundData is a list with three components: cfList, tmList and bM. cfList is list of 44 numeric vectors (the cash flows). tmList is a list of 44 character vectors (the payment dates) formatted as YYYY-MM-DD. bM is a numeric vector with 44 elements (the dirty prices of the bonds).

Details

All prices are as of 31 May 2010. See chapter 14 in Gilli et al. (2011).

Source

The data was obtained from http://www.deutsche-finanzagentur.de . The data is also freely available from the website of the Bundesbank http://www.bundesbank.de .

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

Examples

```
bundData
str(bundData)

## get ISINs of bonds
names(bundData$cfList)

## get a specific bond
thisBond <- "DE0001135358"
data.frame(dates = as.Date(bundData$tmList[[thisBond]]),
           payments = bundData$cfList[[thisBond]])
```

---

bundFuture *Theoretical Valuation of Euro Bund Future*

Description

Compute theoretical prices of bund future.

Usage

```
bundFuture(clean, coupon, trade.date,
           expiry.date, last.coupon.date,
           r, cf)

bundFutureImpliedRate(future, clean, coupon,
```

```
                    trade.date, expiry.date,
                    last.coupon.date, cf)
```

Arguments

clean	numeric: clean prices of CTD
future	numeric: price of future
coupon	numeric
trade.date	Date or character in format YYYY-MM-DD
expiry.date	Date or character in format YYYY-MM-DD
last.coupon.date	Date or character in format YYYY-MM-DD
r	numeric: 0.01
cf	numeric: conversion factor of CTD

Details

bundFuture computes the theoretical prices of the Bund Future, given the prices of the cheapest-to-deliver eligible government bond.

bundFutureImpliedRate computes the implied refinancing rate.

Value

numeric

Author(s)

Enrico Schumann

Examples

```
## Bund-Future with expiry Sep 2017
## CTD: DE0001102408 -- 0%, 15 Aug 2026
##
## On 21 August 2017, the CTD traded (clean) at 97.769
## the FGBL Sep 2017 closed at 164.44.

bundFuture(clean = 97.769,                  ## DE0001102408
           coupon = 0,
           trade.date = "2017-8-21",
           expiry.date = "2017-09-07",      ## Bund expiry
           last.coupon.date = "2017-08-15", ## last co
           r = -0.0037,
           cf = 0.594455)   ## conversion factor (from Eurex website)

bundFutureImpliedRate(future = 164.44,
                      clean = 97.769,
                      coupon = 0,
                      trade.date = "2017-8-21",
```

```
                    expiry.date = "2017-09-07",
                    last.coupon.date = "2017-08-15",
                    cf = 0.594455)
```

---

callCF *Price a Plain-Vanilla Call with the Characteristic Function*

Description

Price a European plain-vanilla call with the characteric function.

Usage

```
callCF(cf, S, X, tau, r, q = 0, ...,
       implVol = FALSE, uniroot.control = list(), uniroot.info = FALSE)
cfBSM(om, S, tau, r, q, v)
cfMerton(om, S, tau, r, q, v, lambda, muJ, vJ)
cfBates(om, S, tau, r, q, v0, vT, rho, k, sigma, lambda, muJ, vJ)
cfHeston(om, S, tau, r, q, v0, vT, rho, k, sigma)
cfVG(om, S, tau, r, q, nu, theta, sigma)
```

Arguments

cf	characteristic function
S	spot
X	strike
tau	time to maturity
r	the interest rate
q	the dividend rate
...	arguments passed to the characteristic function
implVol	logical: compute implied vol?
uniroot.control	A list. If there are elements named interval, tol or maxiter, these are passed to uniroot. Any other elements of the list are ignored.
uniroot.info	logical; default is FALSE. If TRUE, the function will return the information returned by uniroot. See paragraph Value below.
om	a (usually complex) argument
v0	a numeric vector of length one
vT	a numeric vector of length one
v	a numeric vector of length one
rho	a numeric vector of length one
k	a numeric vector of length one
sigma	a numeric vector of length one

lambda	a numeric vector of length one
muJ	a numeric vector of length one
vJ	a numeric vector of length one
nu	a numeric vector of length one
theta	a numeric vector of length one

Details

The function computes the value of a plain vanilla European call under different models, using the representation of Bakshi/Madan. Put values can be computed through put–call parity (see putCallParity).

If implVol is TRUE, the function will compute the implied volatility necessary to obtain the same value under Black–Scholes–Merton. The implied volatility is computed with uniroot from the **stats** package. The default search interval is c(0.00001, 2); it can be changed through uniroot.control.

The function uses variances as inputs (not volatilities).

The function is not vectorised (but see the NMOF Manual for examples of how to efficiently price more than one option at once).

Value

Returns the value of the call (numeric) under the respective model or, if implVol is TRUE, a list of the value and the implied volatility. (If, in addition, uniroot.info is TRUE, the information provided by uniroot is also returned.)

Note

If implVol is TRUE, the function will return a list with elements named value and impliedVol. Prior to version 0.26-3, the first element was named callPrice.

Author(s)

Enrico Schumann

References

Bates, David S. (1996) Jumps and Stochastic Volatility: Exchange Rate Processes Implicit in Deutsche Mark Options. *Review of Financial Studies* **9** (1), 69–107.

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Heston, S.L. (1993) A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bonds and Currency options. *Review of Financial Studies* **6** (2), 327–343.

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

See Also

callHestoncf

Examples

```
S <- 100; X <- 100; tau <- 1
r <- 0.02; q <- 0.08
v0 <- 0.2^2  ## variance, not volatility
vT <- 0.2^2  ## variance, not volatility
v <- vT
rho <- -0.3; k <- .2
sigma <- 0.3

## jump parameters (Merton and Bates)
lambda <- 0.1
muJ <- -0.2
vJ <- 0.1^2

## get Heston price and BSM implied volatility
callHestoncf(S, X, tau, r, q, v0, vT, rho, k, sigma, implVol = FALSE)
callCF(cf = cfHeston, S=S, X=X, tau=tau, r=r, q = q,
       v0 = v0, vT = vT, rho = rho, k = k, sigma = sigma, implVol = FALSE)

## Black-Scholes-Merton
callCF(cf = cfBSM, S=S, X=X, tau = tau, r = r, q = q,
       v = v, implVol = TRUE)

## Bates
callCF(cf = cfBates, S = S, X = X, tau = tau, r = r, q = q,
       v0 = v0, vT = vT, rho = rho, k = k, sigma = sigma,
       lambda = lambda, muJ = muJ, vJ = vJ, implVol = FALSE)

## Merton
callCF(cf = cfMerton, S = S, X = X, tau = tau, r = r, q = q,
       v = v, lambda = lambda, muJ = muJ, vJ = vJ, implVol = FALSE)

## variance gamma
nu <- 0.1; theta <- -0.1; sigma <- 0.15
callCF(cf = cfVG, S = S, X = X, tau = tau, r = r, q = q,
       nu = nu, theta = theta, sigma = sigma, implVol = FALSE)
```

`callHestoncf` *Price of a European Call under the Heston Model*

Description

Computes the price of a European Call under the Heston model (and the equivalent Black–Scholes–Merton volatility)

Usage

```
callHestoncf(S, X, tau, r, q, v0, vT, rho, k, sigma, implVol = FALSE)
```

Arguments

S	current stock price
X	strike price
tau	time to maturity
r	risk-free rate
q	dividend rate
v0	current variance
vT	long-run variance
rho	correlation between spot and variance
k	speed of mean-reversion
sigma	volatility of variance. A value smaller than 0.01 is replaced with 0.01.
implVol	compute equivalent Black–Scholes–Merton volatility? Default is FALSE.

Details

The function computes the value of a plain vanilla European call under the Heston model. Put values can be computed through put–call-parity. If `implVol` is `TRUE`, the function will compute the implied volatility necessary to obtain the same price under Black–Scholes–Merton. The implied volatility is computed with `uniroot` from the **stats** package.

Note that the function takes variances as inputs (not volatilities).

Value

Returns the value of the call (numeric) under the Heston model or, if `implVol` is `TRUE`, a list of the value and the implied volatility.

Note

If `implVol` is `TRUE`, the function will return a list with elements named `value` and `impliedVol`. Prior to version 0.26-3, the first element was named `callPrice`.

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Heston, S.L. (1993) A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bonds and Currency options. *Review of Financial Studies* **6**(2), 327–343.

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

See Also

callCF, EuropeanCall

Examples

```
S <- 100; X <- 100; tau <- 1; r <- 0.02; q <- 0.01
v0  <- 0.2^2  ## variance, not volatility
vT  <- 0.2^2  ## variance, not volatility
rho <- -0.7; k <- 0.2; sigma <- 0.5

## get Heston price and BSM implied volatility
result <- callHestoncf(S = S, X = X, tau = tau, r = r, q = q,
                       v0 = v0, vT = vT, rho = rho, k = k,
                       sigma = sigma, implVol = TRUE)

## Heston price
result[[1L]]

## price BSM with implied volatility
vol <- result[[2L]]
d1 <- (log(S/X) + (r - q + vol^2 / 2)*tau) / (vol*sqrt(tau))
d2 <- d1 - vol*sqrt(tau)
callBSM <- S * exp(-q * tau) * pnorm(d1) -
           X * exp(-r * tau) * pnorm(d2)
callBSM  ## should be (about) the same as result[[1L]]
```

---

callMerton *Price of a European Call under Merton's Jump–Diffusion Model*

Description

Computes the price of a European Call under Merton's jump–diffusion model (and the equivalent Black–Scholes–Merton volatility)

Usage

```
callMerton(S, X, tau, r, q, v, lambda, muJ, vJ, N, implVol = FALSE)
```

Arguments

S	current stock price
X	strike price
tau	time to maturity
r	risk-free rate
q	dividend rate
v	variance
lambda	jump intensity

muJ	mean jump-size
vJ	variance of log jump-size
N	The number of jumps. See Details.
implVol	compute equivalent Black–Scholes–Merton volatility? Default is FALSE.

Details

The function computes the value of a plain-vanilla European call under Merton's jump–diffusion model. Put values can be computed through put–call-parity (see putCallParity). If implVol is TRUE, the function also computes the implied volatility necessary to obtain the same price under Black–Scholes–Merton. The implied volatility is computed with uniroot from the **stats** package.

Note that the function takes variances as inputs (not volatilities).

The number of jumps N typically can be set 10 or 20. (Just try to increase N and see how the results change.)

Value

Returns the value of the call (numeric) or, if implVol is TRUE, a list of the value and the implied volatility.

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Merton, R.C. (1976) Option Pricing when Underlying Stock Returns are Discontinuous. *Journal of Financial Economics* **3**(1–2), 125–144.

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

See Also

callCF, EuropeanCall

Examples

```
S <- 100; X <- 100; tau <- 1
r <- 0.0075; q <- 0.00
v <- 0.2^2
lambda <- 1; muJ <- -0.2; vJ <- 0.6^2
N <- 20

## jumps can make a difference
callMerton(S, X, tau, r, q, v, lambda, muJ, vJ, N, implVol = TRUE)
callCF(cf = cfMerton, S = S, X = X, tau = tau, r = r, q = q,
```

```
       v = v, lambda = lambda, muJ = muJ, vJ = vJ, implVol = TRUE)
vanillaOptionEuropean(S,X,tau,r,q,v, greeks = FALSE)

lambda <- 0 ## no jumps
callMerton(S, X, tau, r, q, v, lambda, muJ, vJ, N, implVol = FALSE)
vanillaOptionEuropean(S,X,tau,r,q,v, greeks = FALSE)

lambda <- 1; muJ <- 0; vJ <- 0.0^2  ## no jumps, either
callMerton(S, X, tau, r, q, v, lambda, muJ, vJ, N, implVol = FALSE)
vanillaOptionEuropean(S,X,tau,r,q,v, greeks = FALSE)
```

`colSubset` *Full-rank Column Subset*

Description

Select a full-rank subset of columns of a matrix.

Usage

```
colSubset(x)
```

Arguments

x	a numeric matrix

Details

Uses `qr`.

Value

A list:

`columns`	indices of columns
`multiplier`	a matrix

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

See Also

repairMatrix

Examples

```
nc <- 3   ## columns
nr <- 10  ## rows
M <- array(rnorm(nr * nc), dim = c(nr, nc))

C <- array(0.5, dim = c(nc, nc))
diag(C) <- 1
M <- M %*% chol(C)
M <- M[ ,c(1,1,1,2,3)]
M

(tmp <- colSubset(M))

C <- cor(M[ ,tmp$columns])
nc <- ncol(C)
nr <- 100
X <- array(rnorm(nr*nc), dim = c(nr, nc))
X <- X %*% chol(C)
X <- X %*% tmp$multiplier
head(X)
cor(X)
```

CPPI *Constant-Proportion Portfolio Insurance*

Description

Simulate constant-proportion portfolio insurance (CPPI) for a given price path.

Usage

```
CPPI(S, multiplier, floor, r, tau = 1, gap = 1)
```

Arguments

S	numeric: price path of risky asset
multiplier	numeric
floor	numeric: a percentage, should be smaller than 1
r	numeric: interest rate (per time period tau)
tau	numeric: time periods
gap	numeric: how often to rebalance. 1 means every timestep, 2 means every second timestep, and so on.

Details

Based on Dietmar Maringer's MATLAB code (function CPPIgap, Listing 9.1).

See Gilli, Maringer and Schumann, 2011, chapter 9.

Value

A list:

V	normalised value (always starts at 1)
C	cushion
B	bond investment
F	floor
E	exposure
N	units of risky asset
S	price path

Author(s)

Original MATLAB code: Dietmar Maringer. R implementation: Enrico Schumann.

References

Chapter 9 of Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2017) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

Examples

```
tau <- 2
S <- gbm(npaths = 1, timesteps = tau*256,
         r = 0.02, v = 0.2^2, tau = tau, S0 = 100)

## rebalancing every day
sol <- CPPI(S, multiplier = 5, floor = 0.9, r = 0.01,
            tau = tau, gap = 1)
par(mfrow = c(3,1), mar = c(3,3,1,1))
plot(0:(length(S)-1), S,     type = "s", main = "stock price")
plot(0:(length(S)-1), sol$V, type = "s", main = "value")
plot(0:(length(S)-1), 100*sol$E/sol$V, type = "s",
     main = "% invested in risky asset")

## rebalancing every 5th day
sol <- CPPI(S, multiplier = 5, floor = 0.9, r = 0.01,
            tau = tau, gap = 5)
par(mfrow = c(3,1), mar = c(3,3,1,1))
plot(0:(length(S)-1), S,     type = "s", main = "stock price")
plot(0:(length(S)-1), sol$V, type = "s", main = "value")
```

```
plot(0:(length(S)-1), 100*sol$E/sol$V, type = "s",
     main = "% invested in risky asset")
```

DEopt *Optimisation with Differential Evolution*

Description

The function implements the standard Differential Evolution algorithm.

Usage

```
DEopt(OF, algo = list(), ...)
```

Arguments

OF	The objective function, to be minimised. See Details.
algo	A list with the settings for algorithm. See Details and Examples.
...	Other pieces of data required to evaluate the objective function. See Details and Examples.

Details

The function implements the standard Differential Evolution (no jittering or other features). Differential Evolution (DE) is a population-based optimisation heuristic proposed by Storn and Price (1997). DE evolves several solutions (collected in the 'population') over a number of iterations ('generations'). In a given generation, new solutions are created and evaluated; better solutions replace inferior ones in the population. Finally, the best solution of the population is returned. See the references for more details on the mechanisms.

To allow for constraints, the evaluation works as follows: after a new solution is created, it is (i) repaired, (ii) evaluated through the objective function, (iii) penalised. Step (ii) is done by a call to OF; steps (i) and (iii) by calls to algo$repair and algo$pen. Step (i) and (iii) are optional, so the respective functions default to NULL. A penalty is a positive number added to the 'clean' objective function value, so it can also be directly written in the OF. Writing a separate penalty function is often clearer; it can be more efficient if either only the objective function or only the penalty function can be vectorised. (Constraints can also be added without these mechanisms. Solutions that violate constraints can, for instance, be mapped to feasible solutions, but without actually changing them. See Maringer and Oyewumi, 2007, for an example.)

Conceptually, DE consists of two loops: one loop across the generations and, in any given generation, one loop across the solutions. DEopt indeed uses, as the default, two loops. But it does not matter in what order the solutions are evaluated (or repaired or penalised), so the second loop can be vectorised. This is controlled by the variables algo$loopOF, algo$loopRepair and algo$loopPen, which all default to TRUE. Examples are given in the vignettes and in the book. The respective algo$loopFun must then be set to FALSE.

All objects that are passed through ... will be passed to the objective function, to the repair function and to the penalty function.

The list algo collects the the settings for the algorithm. Strictly necessary are only min and max (to initialise the population). Here are all possible arguments:

CR probability for crossover. Defaults to 0.9. Using default settings may not be a good idea.

F The step size. Typically a numeric vector of length one; default is 0.5. Using default settings may not be a good idea. (F can also be a vector with different values for each decision variable.)

nP population size. Defaults to 50. Using default settings may not be a good idea.

nG number of generations. Defaults to 300. Using default settings may not be a good idea.

min, max vectors of minimum and maximum parameter values. The vectors min and max are used to determine the dimension of the problem and to randomly initialise the population. Per default, they are no constraints: a solution may well be outside these limits. Only if algo$minmaxConstr is TRUE will the algorithm repair solutions outside the min and max range.

minmaxConstr if TRUE, algo$min and algo$max are considered constraints. Default is FALSE.

pen a penalty function. Default is NULL (no penalty).

initP optional: the initial population. A matrix of size length(algo$min) times algo$nP, or a function that creates such a matrix. If a function, it should take no arguments.

repair a repair function. Default is NULL (no repairing).

loopOF logical. Should the OF be evaluated through a loop? Defaults to TRUE.

loopPen logical. Should the penalty function (if specified) be evaluated through a loop? Defaults to TRUE.

loopRepair logical. Should the repair function (if specified) be evaluated through a loop? Defaults to TRUE.

printDetail If TRUE (the default), information is printed. If an integer i greater then one, information is printed at very ith generation.

printBar If TRUE (the default), a txtProgressBar is printed.

storeF if TRUE (the default), the objective function values for every solution in every generation are stored and returned as matrix Fmat.

storeSolutions default is FALSE. If TRUE, the solutions (ie, decision variables) in every generation are stored and returned as a list P in list xlist (see Value section below). To check, for instance, the solutions at the end of the ith generation, retrieve xlist[[c(1L, i)]]. This will be a matrix of size length(algo$min) times algo$nP. (To be consistent with other functions, xlist is itself a list. In the case of DEopt, it contains just one element.)

Value

A list:

xbest	the solution (the best member of the population), which is a numeric vector
OFvalue	objective function value of best solution
popF	a vector. The objective function values in the final population.
Fmat	if algo$storeF is TRUE, a matrix of size algo$nG times algo$nP containing the objective function values of all solutions over the generations; else NA.
xlist	if algo$storeSolutions is TRUE, a list that contains a list P of matrices and a matrix initP (the initial solution); else NA.
initial.state	the value of .Random.seed when the function was called.

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Maringer, D. and Oyewumi, O. (2007). Index Tracking with Constrained Portfolios. *Intelligent Systems in Accounting, Finance and Management*, **15**(1), pp. 57–71.

Schumann, E. (2012) Remarks on 'A comparison of some heuristic optimization methods'. http://enricoschumann.net/R/remarks.htm

Schumann, E. (2017) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

Storn, R., and Price, K. (1997) Differential Evolution – a Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *Journal of Global Optimization*, **11**(4), pp. 341–359.

See Also

GAopt, PSopt

Examples

```
## Example 1: Trefethen's 100-digit challenge (problem 4)
## http://people.maths.ox.ac.uk/trefethen/hundred.html

OF <- tfTrefethen               ### see ?testFunctions
algo <- list(nP = 50L,          ### population size
             nG = 300L,         ### number of generations
              F = 0.6,          ### step size
             CR = 0.9,          ### prob of crossover
            min = c(-10, -10),  ### range for initial population
            max = c( 10,  10))

sol <- DEopt(OF = OF, algo = algo)
## correct answer: -3.30686864747523
format(sol$OFvalue, digits = 12)
## check convergence of population
sd(sol$popF)
ts.plot(sol$Fmat, xlab = "generations", ylab = "OF")


## Example 2: vectorising the evaluation of the population
OF <- tfRosenbrock     ### see ?testFunctions
size <- 3L             ### define dimension
x <- rep.int(1, size)  ### the known solution ...
OF(x)                  ### ... should give zero

algo <- list(printBar = FALSE,
                   nP = 30L,
                   nG = 300L,
```

```
                    F = 0.6,
                   CR = 0.9,
                  min = rep(-100, size),
                  max = rep( 100, size))

## run DEopt
(t1 <- system.time(sol <- DEopt(OF = OF, algo = algo)))
sol$xbest
sol$OFvalue  ### should be zero (with luck)

## a vectorised Rosenbrock function: works only with a *matrix* x
OF2 <- function(x) {
    n <- dim(x)[1L]
    xi <- x[seq_len(n - 1L), ]
    colSums(100 * (x[2L:n, ] - xi * xi)^2 + (1 - xi)^2)
}

## random solutions (every column of 'x' is one solution)
x <- matrix(rnorm(size * algo$nP), size, algo$nP)
all.equal(OF2(x)[1:3],
          c(OF(x[ ,1L]), OF(x[ ,2L]), OF(x[ ,3L])))

## run DEopt and compare computing time
algo$loopOF <- FALSE
(t2 <- system.time(sol2 <- DEopt(OF = OF2, algo = algo)))
sol2$xbest
sol2$OFvalue       ### should be zero (with luck)
t1[[3L]]/t2[[3L]]  ### speedup
```

`divRatio` *Diversification Ratio*

Description

Compute the diversification ratio of a portfolio.

Usage

```
divRatio(w, var)
```

Arguments

`w`	numeric: a vector of weights
`var`	numeric matrix: the variance–covariance matrix

Details

The function provides an efficient implementation of the diversification ratio, suitable for optimisation.

Value

a numeric vector of length one

Author(s)

Enrico Schumann

References

Yves Choueifaty and Yves Coignard (2008) Toward Maximum Diversification. *Journal of Portfolio Management* **35**(1), 40–51.

See Also

pm, drawdown

Examples

```
na  <- 10     ## number of assets
rho <- 0.5    ## correlation
v_min <- 0.2  ## minimum vol
v_max <- 0.4  ## maximum vol

## set up a covariance matrix S
C <- array(rho, dim = c(na,na))
diag(C) <- 1
vols <- seq(v_min, v_max, length.out = na)
S <- outer(vols, vols) * C

w <- rep(1/na, na)  ## weights
divRatio(w, S)
```

---

drawdown *Drawdown*

Description

Compute the drawdown of a time series.

Usage

```
drawdown(v, relative = TRUE, summary = TRUE)
```

Arguments

v	a price series (a numeric vector)
relative	if TRUE, maximum drawdown is chosen according to percentage losses; else in units of v
summary	if TRUE, provide maximum drawdown and time when it occured; else return drawdown vector

Details

The drawdown at position t of a time series v is the difference between the highest peak that was reached before t and the current value. If the current value represents a new high, the drawdown is zero.

Value

If summary is FALSE, a vector of the same length as v. If summary is TRUE, a list

maximum	maximum drawdown
high	the max of v
high.position	position of high
low	the min of v
low.position	position of low

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*, Chapter 13. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

Examples

```
v <- cumprod(1 + rnorm(20) * 0.02)
drawdown(v)
```

`EuropeanCall` *Computing Prices of European Calls with a Binomial Tree*

Description

Computes the fair value of a European Call with the binomial tree of Cox, Ross and Rubinstein.

Usage

```
EuropeanCall(S0, X, r, tau, sigma, M = 101)
EuropeanCallBE(S0, X, r, tau, sigma, M = 101)
```

Arguments

`S0`	current stock price
`X`	strike price
`r`	risk-free rate
`tau`	time to maturity
`sigma`	volatility
`M`	number of time steps

Details

Prices a European Call with the tree approach of Cox, Ross, Rubinstein.

The algorithm in `EuropeanCallBE` does not construct and traverse a tree, but computes the terminal prices via a binomial expansion (see Higham, 2002, and Chapter 5 in Gilli/Maringer/Schumann, 2011).

Value

Returns the value of the call (`numeric`).

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

M. Gilli and Schumann, E. (2009) Implementing Binomial Trees. COMISEF Working Paper Series No. 008. http://comisef.eu/?q=working_papers

Higham, D. (2002) Nine Ways to Implement the Binomial Method for Option Valuation in MATLAB. *SIAM Review*, **44**(4), pp. 661–677. http://personal.strath.ac.uk/d.j.higham/papers/binom.pdf .

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

See Also

callHestoncf

Examples

```
## price
EuropeanCall(  S0 = 100, X = 100, r = 0.02, tau = 1, sigma = 0.20, M = 50)
EuropeanCallBE(S0 = 100, X = 100, r = 0.02, tau = 1, sigma = 0.20, M = 50)

## a Greek: delta
h <- 1e-8
C1 <- EuropeanCall(S0 = 100 + h, X = 100, r = 0.02, tau = 1,
                   sigma = 0.20, M = 50)
C2 <- EuropeanCall(S0 = 100    , X = 100, r = 0.02, tau = 1,
                   sigma = 0.20, M = 50)
(C1 - C2) / h
```

---

fundData *Mutual Fund Returns*

Description

A matrix of 500 rows (return scenarios) and 200 columns (mutual funds). The elements in the matrix are weekly returns.

Usage

```
fundData
```

Format

A plain numeric matrix.

Details

The scenarios were created with a bootstrapping technique. The data set is only meant to provide example data on which to test algorithms.

Source

Schumann, E. (2010) *Essays on Practical Financial Optimisation*, (chapter 4), PhD thesis, University of Geneva.

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

Examples

```
apply(fundData, 2, summary)
```

GAopt *Optimisation with a Genetic Algorithm*

Description

A simple Genetic Algorithm for minimising a function.

Usage

```
GAopt (OF, algo = list(), ...)
```

Arguments

OF	The objective function, to be minimised. See Details.
algo	A list with the settings for algorithm. See Details and Examples.
...	Other pieces of data required to evaluate the objective function. See Details and Examples.

Details

The function implements a simple Genetic Algorithm (GA). A GA evolves a collection of solutions (the so-called population), all of which are coded as vectors containing only zeros and ones. (In GAopt, solutions are of mode logical.) The algorithm starts with randomly-chosen or user-supplied population and aims to iteratively improve this population by mixing solutions and by switching single bits in solutions, both at random. In each iteration, such randomly-changed solutions are compared with the original population and better solutions replace inferior ones. In GAopt, the population size is kept constant.

GA language: iterations are called generations; new solutions are called offspring or children (and the existing solutions, from which the children are created, are parents); the objective function is called a fitness function; mixing solutions is a crossover; and randomly changing solutions is called mutation. The choice which solutions remain in the population and which ones are discarded is called selection. In GAopt, selection is pairwise: a given child is compared with a given parent; the better of the two is kept. In this way, the best solution is automatically retained in the population.

To allow for constraints, the evaluation works as follows: after new solutions are created, they are (i) repaired, (ii) evaluated through the objective function, (iii) penalised. Step (ii) is done by a call to OF; steps (i) and (iii) by calls to algo$repair and algo$pen. Step (i) and (iii) are optional, so the respective functions default to NULL. A penalty can also be directly written in the OF, since it amounts to a positive number added to the 'clean' objective function value; but a separate function is often clearer. A separate penalty function is advantagous if either only the objective function or only the penalty function can be vectorised.

Conceptually a GA consists of two loops: one loop across the generations and, in any given generation, one loop across the solutions. This is the default, controlled by the variables algo$loopOF,

algo$loopRepair and algo$loopPen, which all default to TRUE. But it does not matter in what order the solutions are evaluated (or repaired or penalised), so the second loop can be vectorised. The respective algo$loopFun must then be set to FALSE. (See also the examples for DEopt and PSopt.)

The evaluation of the objective function in a given generation can even be distributed. For this, an argument algo$methodOF needs to be set; see below for details (and Schumann, 2011, for examples).

All objects that are passed through ... will be passed to the objective function, to the repair function and to the penalty function.

The list algo contains the following items:

nB number of bits per solution. Must be specified.

nP population size. Defaults to 50. Using default settings may not be a good idea.

nG number of iterations ('generations'). Defaults to 300. Using default settings may not be a good idea.

crossover The crossover method. Default is "onePoint"; also possible is "uniform".

prob The probability for switching a single bit. Defaults to 0.01; typically a small number.

pen a penalty function. Default is NULL (no penalty).

repair a repair function. Default is NULL (no repairing).

initP optional: the initial population. A logical matrix of size length(algo$nB) times algo$nP, or a function that creates such a matrix. If a function, it must take no arguments. If mode(mP) is not logical, then storage.mode(mP) will be tried (and a warning will be issued).

loopOF logical. Should the OF be evaluated through a loop? Defaults to TRUE.

loopPen logical. Should the penalty function (if specified) be evaluated through a loop? Defaults to TRUE.

loopRepair logical. Should the repair function (if specified) be evaluated through a loop? Defaults to TRUE.

methodOF loop (the default), vectorised, snow or multicore. Setting vectorised is equivalent to having algo$loopOF set to FALSE (and methodOF overrides loopOF). snow and multicore use functions clusterApply and mclapply, respectively. For snow, an object algo$cl needs to be specified (see below). For multicore, optional arguments can be passed through algo$mc.control (see below).

cl a cluster object or the number of cores. See documentation of package parallel.

mc.control a list of named elements; optional settings for mclapply (for instance, list(mc.set.seed = FALSE))

printDetail If TRUE (the default), information is printed.

printBar If TRUE (the default), a txtProgressBar is printed.

storeF If TRUE (the default), the objective function values for every solution in every generation are stored and returned as matrix Fmat.

storeSolutions If TRUE, the solutions (ie, binary strings) in every generation are stored and returned as a list P in list xlist (see Value section below). To check, for instance, the solutions at the end of the ith generation, retrieve xlist[[c(1L, i)]]. This will be a matrix of size algo$nB times algo$nP.

Value

A list:

xbest	the solution (the best member of the population)
OFvalue	objective function value of best solution
popF	a vector. The objective function values in the final population.
Fmat	if algo$storeF is TRUE, a matrix of size algo$nG times algo$nP containing the objective function values of all solutions over the generations; else NA
xlist	if algo$storeSolutions is TRUE, a list that contains a list P of matrices and a matrix initP (the initial solution); else NA.
initial.state	the value of .Random.seed when the function was called.

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2017) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

See Also

DEopt, PSopt

Examples

```
## a *very* simple problem (why?):
## match a binary (logical) string y

size <- 20L  ### the length of the string
OF <- function(x, y) sum(x != y)
y <- runif(size) > 0.5
x <- runif(size) > 0.5
OF(y, y)     ### the optimum value is zero
OF(x, y)
algo <- list(nB = size, nP = 20L, nG = 100L, prob = 0.002,
             printBar = TRUE)
sol <- GAopt(OF, algo = algo, y = y)

## show differences (if any: marked by a '^')
cat(as.integer(y), "\n", as.integer(sol$xbest), "\n",
    ifelse(y == sol$xbest , " ", "^"), "\n", sep = "")

algo$nP <- 3L  ### that shouldn't work so well
sol2 <- GAopt(OF, algo = algo, y = y)
```

```
## show differences (if any: marked by a '^')
cat(as.integer(y), "\n", as.integer(sol2$xbest), "\n",
    ifelse(y == sol2$xbest , " ", "^"), "\n", sep = "")
```

---

gridSearch *Grid Search*

Description

Evaluate a function for a given list of possible arguments.

Usage

```
gridSearch(fun, levels, ..., lower, upper, npar = 1L, n = 5L,
           printDetail = TRUE,
           method = NULL,
           mc.control = list(), cl = NULL,
           keepNames = FALSE, asList = FALSE)
```

Arguments

fun	a function of the form fun(x, ...), with x being a numeric vector or a list of numeric vectors (of length one)
levels	a list of levels for the arguments (see Examples)
...	objects passed to fun
lower	a numeric vector. Ignored if levels are explicitly specified.
upper	a numeric vector. Ignored if levels are explicitly specified.
npar	the number of parameters. Must be supplied if lower and upper are to be expanded; see Details. Ignored when levels are explicitly specified, or when lower/upper are used and at least one has lenght greater than one. See Examples.
n	the number of levels. Default is 5. Ignored if levels are explicitly specified.
printDetail	print information on the number of objective function evaluations
method	can be loop (the default), multicore or snow. See Details.
mc.control	a list containing settings that will be passed to mclapply if method is multicore. Must be a list of named elements; see the documentation of mclapply in **parallel**.
cl	default is NULL. If method snow is used, this must be a cluster object or an integer (the number of cores).
keepNames	logical: should the names of levels be kept?
asList	does fun expect a list? Default is FALSE.

Details

A grid search can be used to find 'good' parameter values for a function. In principle, a grid search has an obvious deficiency: as the length of `x` (the first argument to `fun`) increases, the number of necessary function evaluations grows exponentially. Note that `gridSearch` will not warn about an unreasonable number of function evaluations, but if `printDetail` is `TRUE` it will print the required number of function evaluations.

In practice, grid search is often better than its reputation. If a function takes only a few parameters, it is often a reasonable approach to find 'good' parameter values.

The function uses the mechanism of `expand.grid` to create the list of parameter combinations for which `fun` is evaluated; it calls `lapply` to evaluate `fun` if `method == "loop"` (the default).

If `method` is `multicore`, then function `mclapply` from package **parallel** is used. Further settings for `mclapply` can be passed through the list `mc.control`. If `multicore` is chosen but the functionality is not available, then `method` will be set to `loop` and a warning is issued. If `method == "snow"`, the function `clusterApply` from package **parallel** is used. In this case, the argument `cl` must either be a cluster object (see the documentation of `clusterApply`) or an integer. If an integer, a cluster will be set up via `makeCluster(c(rep("localhost", cl)), type = "SOCK")` (and `stopCluster` is called when the function is exited). If `snow` is chosen but not available or `cl` is not specified, then `method` will be set to `loop` and a warning is issued.

Value

A list.

`minfun`	the minimum of `fun`.
`minlevels`	the levels that give this minimum.
`values`	a list. All the function values of `fun`.
`levels`	a list. All the levels for which `fun` was evaluated.

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

Examples

```
testFun <- function(x)
    x[1L] + x[2L]^2

sol <- gridSearch(fun = testFun, levels = list(1:2, c(2, 3, 5)))
sol$minfun
sol$minlevels
```

```
## specify all levels
levels <- list(a = 1:2, b = 1:3)
res <- gridSearch(testFun, levels)
res$minfun
sol$minlevels

## specify lower, upper and npar
lower <- 1; upper <- 3; npar <- 2
res <- gridSearch(testFun, lower = lower, upper = upper, npar = npar)
res$minfun
sol$minlevels

## specify lower, upper, npar and n
lower <- 1; upper <- 3; npar <- 2; n <- 4
res <- gridSearch(testFun, lower = lower, upper = upper, npar = npar, n = n)
res$minfun
sol$minlevels

## specify lower, upper and n
lower <- c(1,1); upper <- c(3,3); n <- 4
res <- gridSearch(testFun, lower = lower, upper = upper, n = n)
res$minfun
sol$minlevels

## specify lower, upper (auto-expanded) and n
lower <- c(1,1); upper <- 3; n <- 4
res <- gridSearch(testFun, lower = lower, upper = upper, n = n)
res$minfun
sol$minlevels
```

`LS.info` *Local-Search Information*

Description

The function can be called from the objective and neighbourhood function during a run of `LSopt`; it provides information such as the current iteration.

Usage

```
LS.info(n = 0L)
```

Arguments

`n`	generational offset; see Details.

Details

This function is still experimental.

The function can be called in the neighbourhood function or the objective function during a run of LSopt. It evaluates to a list with the state of the optimisation run, such as the current iteration.

LS.info relies on parent.frame to retrieve its information. If the function is called within another function in the neighbourhood or objective function, the argument n needs to be increased.

Value

A list

iteration	current iteration
step	same as 'iteration'

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

See Also

LSopt, TA.info

Examples

```
## MINIMAL EXAMPLE for LSopt

## objective function evaluates to a constant
fun <- function(x)
    0

## neighbourhood function does not even change the solution,
## but it reports information
nb <- function(x) {
    tmp <- LS.info()
    cat("current iteration ", tmp$iteration, "\n")
    x
}

## run LS
algo <- list(nS = 5,
             x0 = rep(0, 5),
             neighbour = nb,
             printBar = FALSE)
```

```
ignore <- LSopt(fun, algo)
```

LSopt *Stochastic Local Search*

Description

Performs a simple stochastic Local Search.

Usage

```
LSopt(OF, algo = list(), ...)
```

Arguments

OF	The objective function, to be minimised. Its first argument needs to be a solution; ... arguments are also passed.
algo	List of settings. See Details.
...	Other variables to be passed to the objective function and to the neighbourhood function. See Details.

Details

Local Search (LS) changes an initial solution for a number of times, accepting only such changes that lead to an improvement in solution quality (as measured by the objective function OF). More specifically, in each iteration, a current solution xc is changed through a function algo$neighbour. This function takes xc as an argument and returns a new solution xn. If xn is not worse than xc, ie, if OF(xn,...)<=OF(xc,...), then xn replaces xc.

The list algo contains the following items:

nS The number of steps. The default is 1000; but this setting depends very much on the problem.

x0 The initial solution. This can be a function; it will then be called once without arguments to compute an initial solution, ie, x0 <- algo$x0(). This can be useful when LSopt is called in a loop of restarts and each restart is to have its own starting value.

neighbour The neighbourhood function, called as neighbour(x, ...). Its first argument must be a solution x; it must return a changed solution.

printDetail If TRUE (the default), information is printed. If an integer i greater then one, information is printed at very ith step.

printBar If TRUE (the default), a txtProgressBar (from package **utils**) is printed). The progress bar is not shown if printDetail is an integer greater than 1.

storeF if TRUE (the default), the objective function values for every solution in every generation are stored and returned as matrix Fmat.

storeSolutions default is FALSE. If TRUE, the solutions (ie, decision variables) in every generation are stored and returned in list xlist (see Value section below). To check, for instance, the current solution at the end of the ith generation, retrieve xlist[[c(2L, i)]].

`OF.target` Numeric; when specified, the algorithm will stop when an objective-function value as low as `OF.target` (or lower) is achieved. This is useful when an optimal objective-function value is known: the algorithm will then stop and not waste time searching for a better solution.

At the minimum, `algo` needs to contain an initial solution `x0` and a `neighbour` function.

`LS` works on solutions through the functions `neighbour` and `OF`, which are specified by the user. Thus, a solution need not be a numeric vector, but can be any other data structure as well (eg, a list or a matrix).

To run silently (except for warnings and errors), `algo$printDetail` and `algo$printBar` must be `FALSE`.

Value

A list:

`xbest`	best solution found.
`OFvalue`	objective function value associated with best solution.
`Fmat`	a matrix with two columns. `Fmat[ ,1L]` contains the proposed solution over all iterations; `Fmat[ ,2L]` contains the accepted solutions.
`xlist`	if `algo$storeSolutions` is `TRUE`, a list; else `NA`. Contains the neighbour solutions at a given iteration (`xn`) and the current solutions (`xc`). Example: `Fmat[i, 2L]` is the objective function value associated with `xlist[[c(2L, i)]]`.
`initial.state`	the value of `.Random.seed` when the function was called.

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

See Also

`TAopt`, `restartOpt`

Examples

```
## Aim: find the columns of X that, when summed, give y

## random data set
nc <- 25L          ## number of columns in data set
nr <- 5L           ## number of rows in data set
howManyCols <- 5L  ## length of true solution
X <- array(runif(nr*nc), dim = c(nr, nc))
```

```
xTRUE <- sample(1L:nc, howManyCols)
Xt <- X[ , xTRUE, drop = FALSE]
y <- rowSums(Xt)

## a random solution x0 ...
makeRandomSol <- function(nc) {
    ii <- sample.int(nc, sample.int(nc, 1L))
    x0 <- logical(nc); x0[ii] <- TRUE
    x0
}
x0 <- makeRandomSol(nc)

## ... but probably not a good one
sum(y - rowSums(X[ , xTRUE, drop = FALSE])) ## should be 0
sum(y - rowSums(X[ , x0, drop = FALSE]))

## a neighbourhood function: switch n elements in solution
neighbour <- function(xc, Data) {
    xn <- xc
    p <- sample.int(Data$nc, Data$n)
    xn[p] <- !xn[p]
    if (sum(xn) < 1L)
        xn <- xc
    xn
}

## a greedy neighbourhood function
neighbourG <- function(xc, Data) {
    of <- function(x)
        abs(sum(Data$y - rowSums(Data$X[ ,x, drop = FALSE])))
    xbest <- xc
    Fxbest <- of(xbest)
    for (i in 1L:Data$nc) {
        xn <- xc; p <- i
        xn[p] <- !xn[p]
        if (sum(xn) >= 1L) {
            Fxn <- of(xn)
            if (Fxn < Fxbest) {
                xbest <- xn
                Fxbest <- Fxn
            }
        }
    }
    xbest
}

## an objective function
OF <- function(xn, Data)
    abs(sum(Data$y - rowSums(Data$X[ ,xn, drop = FALSE])))


## (1) GREEDY SEARCH
## note: this could be done in a simpler fashion, but the
```

```
##       redundancies/overhead here are small, and the example is to
##       show how LSopt can be used for such a search
Data <- list(X = X, y = y, nc = nc, nr = nr, n = 1L)
algo <- list(nS = 500L, neighbour = neighbourG, x0 = x0,
             printBar = FALSE, printDetail = FALSE)
solG <- LSopt(OF, algo = algo, Data = Data)

## after how many iterations did we stop?
iterG <- min(which(solG$Fmat[ ,2L] == solG$OFvalue))
solG$OFvalue  ## the true solution has OF-value 0

## (2) LOCAL SEARCH
algo$neighbour <- neighbour
solLS <- LSopt(OF, algo = algo, Data = Data)
iterLS <- min(which(solLS$Fmat[ ,2L] == solLS$OFvalue))
solLS$OFvalue  ## the true solution has OF-value 0

## (3) *Threshold Accepting*
algo$nT <- 10L
algo$nS <- ceiling(algo$nS/algo$nT)
algo$q <- 0.99
solTA <- TAopt(OF, algo = algo, Data = Data)
iterTA <- min(which(solTA$Fmat[ ,2L] == solTA$OFvalue))
solTA$OFvalue  ## the true solution has OF-value 0

## look at the solution
all <- sort(unique(c(which(solTA$xbest),
                     which(solLS$xbest),
                     which(solG$xbest),
                     xTRUE)))
ta <- ls <- greedy <- true <- character(length(all))
true[  match(xTRUE, all)] <- "o"
greedy[match(which(solG$xbest),  all)] <- "o"
ls[    match(which(solLS$xbest), all)] <- "o"
ta[    match(which(solTA$xbest), all)] <- "o"
data.frame(true = true, greedy = greedy, LS = ls , TA = ta,
           row.names=all)

## plot results
par(ylog = TRUE, mar = c(5,5,1,6), las = 1)
plot(solTA$Fmat[seq_len(iterTA) ,2L],type = "l", log = "y",
     ylim = c(1e-4,
              max(pretty(c(solG$Fmat,solLS$Fmat,solTA$Fmat)))),
     xlab = "iterations", ylab = "OF value", col = grey(0.5))
lines(cummin(solTA$Fmat[seq_len(iterTA), 2L]), type = "l")
lines(solG$Fmat[ seq_len(iterG),  2L], type = "p", col = "blue")
lines(solLS$Fmat[seq_len(iterLS), 2L], type = "l", col = "goldenrod3")
legend(x = "bottomleft",
       legend = c("TA best solution", "TA current solution",
                  "Greedy", "LS current/best solution"),
       lty = c(1,1,0,1),
       col = c("black",grey(0.5),"blue","goldenrod2"),
       pch = c(NA,NA,21,NA))
```

```
axis(4, at = c(solG$OFvalue, solLS$OFvalue, solTA$OFvalue),
        labels = NULL, las = 1)
lines(x = c(iterG, par()$usr[2L]), y = rep(solG$OFvalue,2),
      col = "blue", lty = 3)
lines(x = c(iterTA, par()$usr[2L]), y = rep(solTA$OFvalue,2),
      col = "black", lty = 3)
lines(x = c(iterLS, par()$usr[2L]), y = rep(solLS$OFvalue,2),
      col = "goldenrod3", lty = 3)
```

MA *Simple Moving Average*

Description

The function computes a moving average of a vector.

Usage

```
MA(y, order, pad = NULL)
```

Arguments

y	a numeric vector
order	An integer. The order of the moving average. The function is defined such that order one returns y (see Examples).
pad	Defaults to NULL. If not NULL, all elements of the returned moving average with position smaller than order are replaced by the value of pad. Sensible values may be NA or 0.

Value

Returns a vector of length length(y).

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

Examples

```
MA(1:10, 3)
MA(1:10, 3, pad = NA)

y <- seq(1, 4, by = 0.3)
z <- MA(y, 1)
all(y == z)      ### (typically) FALSE
all.equal(y, z)  ### should be TRUE

## 'Relative strength index'
rsi <- function(y, t) {
    y <- diff(y)
    ups   <- y + abs(y)
    downs <- y - abs(y)
    RS <- -MA(ups, t) / MA(downs, t)
    RS/(1 + RS)
}
x <- cumprod(c(100, 1 + rnorm(100, sd = 0.01)))
par(mfrow = c(2,1))
plot(x, type = "l")
plot(rsi(x, 14), ylim = c(0,1), type = "l")
```

---

mc *Option Pricing via Monte-Carlo Simulation*

Description

Functions to calculate the theoretical prices of options through simulation.

Usage

```
gbm(npaths, timesteps, r, v, tau, S0)
gbb(npaths, timesteps, S0, ST, v, tau)
```

Arguments

npaths	the number of paths
timesteps	timesteps per path
r	the mean per unit of time
v	the variance per unit of time
tau	time
S0	initial value
ST	final value of Brownian bridge

Details

gbm generates sample paths of geometric Brownian motion.

gbb generates sample paths of a Brownian bridge by first creating paths of Brownian motion W from time 0 to time T, with W_0 equal to zero. Then, at each t, it subtracts t/T * W_T and adds S0*(1-t/T)+ST*(t/T).

Value

A matrix of sample paths; each column contains the price path of an asset. Even with only a single time-step, the matrix will have two rows (the first row is S0).

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

See Also

vanillaOptionEuropean

Examples

```
## price a European option
## ... parameters
npaths <- 5000   ## increase number to get more precise results
timesteps <- 1
S0   <- 100
ST   <- 100
tau  <- 1
r <- 0.01
v   <- 0.25^2

## ... create paths
paths <- gbm(npaths, timesteps, r, v, tau, S0 = S0)

## ... a helper function
mc <- function(paths, payoff, ...)
    payoff(paths, ...)

## ... a payoff function (European call)
payoff <- function(paths, X, r, tau)
    exp(-r * tau) * mean(pmax(paths[NROW(paths), ] - X, 0))

## ... compute and check
```

```
mc(paths, payoff, X = 100, r = r, tau = tau)
vanillaOptionEuropean(S0, X = 100, tau = tau, r = r, v = v)$value


## compute delta via forward difference
## (see Gilli/Maringer/Schumann, ch. 9)
h <- 1e-6                 ## a small number
rnorm(1)                  ## make sure RNG is initialised
rnd.seed <- .Random.seed  ## store current seed
paths1 <- gbm(npaths, timesteps, r, v, tau, S0 = S0)
.Random.seed <- rnd.seed
paths2 <- gbm(npaths, timesteps, r, v, tau, S0 = S0 + h)

delta.mc <- (mc(paths2, payoff, X = 100, r = r, tau = tau)-
             mc(paths1, payoff, X = 100, r = r, tau = tau))/h
delta <- vanillaOptionEuropean(S0, X = 100, tau = tau,
                               r = r, v = v)$delta
delta.mc - delta




## a fanplot
steps <- 100
paths <- results <- gbm(1000, steps, r = 0, v = 0.2^2,
                        tau = 1, S0 = 100)

levels <- seq(0.01, 0.49, length.out = 20)
greys  <- seq(0.9,  0.50, length.out = length(levels))

## start with an empty plot ...
plot(0:steps, rep(100, steps+1), ylim = range(paths),
     xlab = "", ylab = "", lty = 0, type = "l")

## ... and add polygons
for (level in levels) {

    l <- apply(paths, 1, quantile, level)
    u <- apply(paths, 1, quantile, 1 - level)
    col <- grey(greys[level == levels])
    polygon(c(0:steps, steps:0), c(l, rev(u)),
            col = col, border = NA)

    ## add border lines
    ## lines(0:steps, l, col = grey(0.4))
    ## lines(0:steps, u, col = grey(0.4))
}
```

`minvar` *Minimum-Variance Portfolios*

Description

Compute minimum-variance portfolios, subject to lower and upper bounds on weights.

Usage

```
minvar(var, wmin = 0, wmax = 1, method = "qp")
```

Arguments

var	the covariance matrix: a numeric (real), symmetric matrix
wmin	numeric: a lower bound on weights. May also be a vector that holds specific bounds for each asset.
wmax	numeric: an upper bound on weights. May also be a vector that holds specific bounds for each asset.
method	character. Currently, only "qp" is supported.

Details

The function uses solve.QP from package **quadprog** package. Because of the algorithm that solve.QP uses, var has to be positive definit (i.e. must be of full rank).

Value

a numeric vector (the portfolio weights) with an attribute variance (the portfolio's variance)

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. https://www.elsevier.com/books/numerical-methods-and-optimization-in-finance/gilli/978-0-12-375662-6

Schumann, E. (2017) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

See Also

TAopt

Examples

```
## variance-covariance matrix from daily returns, 1 Jan 2014 -- 31 Dec 2013, of
## cleaned data set at http://enricoschumann.net/data/gilli_accuracy.html

if (requireNamespace("quadprog")) {

   var <- structure(c(0.000988087100677907, -0.0000179669410403153, 0.000368923882626859,
```

```
                      0.000208303611101873, 0.000262742052359594, -0.0000179669410403153,
                       0.00171852167358765, 0.0000857467457561209, 0.0000215059246610556,
                      0.0000283532159921211, 0.000368923882626859, 0.0000857467457561209,
                        0.00075871953281751, 0.000194002299424151, 0.000188824454515841,
                       0.000208303611101873, 0.0000215059246610556, 0.000194002299424151,
                        0.000265780633005374, 0.000132611196599808, 0.000262742052359594,
                       0.0000283532159921211, 0.000188824454515841, 0.000132611196599808,
                        0.00025948420130626),
                      .Dim = c(5L, 5L),
                    .Dimnames = list(c("CBK.DE", "VOW.DE", "CON.DE", "LIN.DE", "MUV2.DE"),
                                     c("CBK.DE", "VOW.DE", "CON.DE", "LIN.DE", "MUV2.DE")))

## 
##            CBK.DE     VOW.DE    CON.DE    LIN.DE   MUV2.DE
## CBK.DE   0.000988 -0.0000180 0.0003689 0.0002083 0.0002627
## VOW.DE  -0.000018  0.0017185 0.0000857 0.0000215 0.0000284
## CON.DE   0.000369  0.0000857 0.0007587 0.0001940 0.0001888
## LIN.DE   0.000208  0.0000215 0.0001940 0.0002658 0.0001326
## MUV2.DE  0.000263  0.0000284 0.0001888 0.0001326 0.0002595
## 

minvar(var, wmin = 0, wmax = 0.5)

minvar(var,
       wmin = c(0.1,0,0,0,0), ## enforce at least 10% weight in CBK.DE
       wmax = 0.5)

minvar(var, wmin = -Inf, wmax = Inf)   ## no bounds
## [1] -0.0467  0.0900  0.0117  0.4534  0.4916

minvar(var, wmin = -Inf, wmax = 0.45)  ## no lower bounds
## [1] -0.0284  0.0977  0.0307  0.4500  0.4500

minvar(var, wmin =  0.1, wmax = Inf)   ## no upper bounds
## [1] 0.100 0.100 0.100 0.363 0.337

}
```

NS *Zero Rates for Nelson–Siegel–Svensson Model*

Description

Compute zero yields for Nelson–Siegel (NS)/Nelson–Siegel–Svensson (NSS) model.

Usage

```
NS(param, tm)
NSS(param, tm)
```

Arguments

`param`	a vector. For NS: β_1, β_2, β_3, λ. For NSS: a vector: β_1, β_2, β_3, β_4, λ_1, λ_2.
`tm`	a vector of maturities

Details

See Chapter 14 in Gilli/Maringer/Schumann (2011).

Maturities (`tm`) need to be given in time (not dates).

Value

The function returns a vector of length `length(tm)`.

Author(s)

Enrico Schumann

References

Gilli, M. and Grosse, S. and Schumann, E. (2010) Calibrating the Nelson-Siegel-Svensson model, COMISEF Working Paper Series No. 031. http://comisef.eu/files/wps031.pdf

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Gilli, M. and Schumann, E. (2010) A Note on 'Good' Starting Values in Numerical Optimisation, COMISEF Working Paper Series No. 044. http://comisef.eu/files/wps044.pdf

Nelson, C.R. and Siegel, A.F. (1987) Parsimonious Modeling of Yield Curves. *Journal of Business*, **60**(4), pp. 473–489.

Schumann, E. (2017) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

Svensson, L.E. (1994) Estimating and Interpreting Forward Interest Rates: Sweden 1992–1994. IMF Working Paper 94/114.

See Also

`NSf`, `NSSf`

Examples

```
tm <- c(c(1, 3, 6, 9) / 12, 1:10)  ## in years
param <- c(6, 3, 8, 1)
yM <- NS(param, tm)
plot(tm, yM, xlab = "maturity in years",
             ylab = "yield in percent")

param <- c(6, 3, 5, -5, 1, 3)
yM <- NSS(param, tm)
plot(tm, yM, xlab = "maturity in years",
             ylab = "yield in percent")
```

```
## Not run:
## get Bliss/Diebold/Li data (used in some of the papers in References)
u <- url("http://www.ssc.upenn.edu/~fdiebold/papers/paper49/FBFITTED.txt")
open(u); BliDiLi <- scan(u, skip = 14); close(u)
mat <- NULL
for (i in 1:372)
    mat <- rbind(mat,BliDiLi[(19*(i-1)+1):(19*(i-1)+19)])
mats  <- c(1,3,6,9,12,15,18,21,24,30,36,48,60,72,84,96,108,120)/12

## the obligatory perspective plot
persp(x = mat[,1], y = mats, mat[ ,-1L],
      phi = 30, theta = 30, ticktype = "detailed",
      xlab = "time",
      ylab = "time to maturity in years",
      zlab = "zero rates in %")

## End(Not run)
```

NSf *Factor Loadings for Nelson–Siegel and Nelson–Siegel–Svensson*

Description

Computes the factor loadings for Nelson–Siegel (NS) and Nelson–Siegel–Svensson (NSS) model for given `lambda` values.

Usage

```
NSf(lambda, tm)
NSSf(lambda1, lambda2, tm)
```

Arguments

`lambda`	the λ parameter of the NS model (a scalar)
`lambda1`	the λ_1 parameter of the NSS model (a scalar)
`lambda2`	the λ_2 parameter of the NSS model (a scalar)
`tm`	a numeric vector with times-to-payment/maturity

Details

The function computes the factor loadings for given λ parameters. Checking the correlation between these factor loadings can help to set reasonable λ values for the NS/NSS models.

Value

For NS, a matrix with `length(tm)` rows and three columns. For NSS, a matrix with `length(tm)` rows and four columns.

Author(s)

Enrico Schumann

References

Gilli, M. and Grosse, S. and Schumann, E. (2010) Calibrating the Nelson-Siegel-Svensson model, COMISEF Working Paper Series No. 031. http://comisef.eu/files/wps031.pdf

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Gilli, M. and Schumann, E. (2010) A Note on 'Good' Starting Values in Numerical Optimisation, COMISEF Working Paper Series No. 044. http://comisef.eu/files/wps044.pdf

Nelson, C.R. and Siegel, A.F. (1987) Parsimonious Modeling of Yield Curves. *Journal of Business*, **60**(4), pp. 473–489.

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

Svensson, L.E. (1994) Estimating and Interpreting Forward Interest Rates: Sweden 1992–1994. IMF Working Paper 94/114.

See Also

NS, NSS

Examples

```
## Nelson-Siegel
cor(NSf(lambda = 6, tm = 1:10)[-1L, -1L])

## Nelson-Siegel-Svensson
cor(NSSf(lambda1 = 1, lambda2 = 5, tm = 1:10)[-1L, -1L])
cor(NSSf(lambda1 = 4, lambda2 = 9, tm = 1:10)[-1L, -1L])
```

---

optionData *Option Data*

Description

Closing prices of DAX index options as of 2012-02-10.

Usage

```
optionData
```

Format

`optionData` is a list with six components:

`pricesCall` a matrix of size 124 times 10. The rows are the strikes; each column belongs to one expiry date.

`pricesPut` a matrix of size 124 times 10

`index` The DAX index (spot).

`future` The available future settlement prices.

`Euribor` Euribor rates.

`NSSpar` Paramaters for German government bond yields, as estimated by the Bundesbank.

Details

Settlement prices for EUREX options are computed at 17:30, Frankfurt Time, even though trading continues until 22:00.

Source

The data was obtained from several websites: close prices of EUREX products were collected from `www.eurexchange.com` ; Euribor rates and the parameters of the Nelson-Siegel-Svensson can be found at `www.bundesbank.de`

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. `http://www.elsevierdirect.com/product.jsp?isbn=9780123756626`

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). `http://enricoschumann.net/NMOF.htm#NMOFmanual`

Examples

```
str(optionData)
NSS(optionData$NSSpar, 1:10)
```

---

pm *Partial Moments*

Description

Compute partial moments.

Usage

```
pm(x, xp = 2, threshold = 0, lower = TRUE,
   normalise = FALSE, na.rm = FALSE)
```

Arguments

x	a numeric vector or a matrix
xp	exponent
threshold	a numeric vector of length one
lower	logical
normalise	logical
na.rm	logical

Details

For a vector x of length n, partial moments are computed as follows:

$$\text{upper partial moment} = \frac{1}{n}\sum_{x>t}(x-t)^e$$

$$\text{lower partial moment} = \frac{1}{n}\sum_{x<t}(t-x)^e$$

The threshold is denoted t, the exponent xp is labelled e.

If normalise is TRUE, the result is raised to 1/xp. If x is a matrix, the function will compute the partial moments column-wise.

See Gilli, Maringer and Schumann (2011), Section 13.3.

Value

numeric

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*, Chapter 13. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2017) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

Examples

```
pm(x <- rnorm(100), 2)
var(x)/2

pm(x, 2, normalise = TRUE)
sqrt(var(x)/2)
```

PSopt *Particle Swarm Optimisation*

Description

The function implements Particle Swarm Optimisation.

Usage

```
PSopt(OF, algo = list(), ...)
```

Arguments

OF	the objective function to be minimised. See Details.
algo	a list with the settings for algorithm. See Details and Examples.
...	pieces of data required to evaluate the objective function. See Details.

Details

The function implements Particle Swarm Optimisation (PS); see the references for details on the implementation. PS is a population-based optimisation heuristic. It develops several solutions (a 'population') over a number of iterations. PS is directly applicable to continuous problems since the population is stored in real-valued vectors. In each iteration, a solution is updated by adding another vector called velocity. Think of a solution as a position in the search space, and of velocity as the direction into which this solution moves. Velocity changes over the course of the optimization: it is biased towards the best solution found by the particular solution and the best overall solution. The algorithm stops after a fixed number of iterations.

To allow for constraints, the evaluation works as follows: after a new solution is created, it is (i) repaired, (ii) evaluated through the objective function, (iii) penalised. Step (ii) is done by a call to OF; steps (i) and (iii) by calls to algo$repair and algo$pen. Step (i) and (iii) are optional, so the respective functions default to NULL. A penalty can also be directly written in the OF, since it amounts to a positive number added to the 'clean' objective function value. It can be advantageous to write a separate penalty function if either only the objective function or only the penalty function can be vectorised. (Constraints can also be added without these mechanisms. Solutions that violate constraints can, for instance, be mapped to feasible solutions, but without actually changing them. See Maringer and Oyewumi, 2007, for an example with Differential Evolution.)

Conceptually, PS consists of two loops: one loop across the iterations and, in any given generation, one loop across the solutions. This is the default, controlled by the variables algo$loopOF, algo$loopRepair, algo$loopPen and loopChangeV which all default to TRUE. But it does not matter in what order the solutions are evaluated (or repaired or penalised), so the second loop can be vectorised. Examples are given in the vignettes and in the book. The respective algo$loopFun must then be set to FALSE.

The objective function, the repair function and and the penalty function will be called as fun(solution, ...).

The list algo contains the following items:

nP population size. Defaults to 100. Using default settings may not be a good idea.

nG number of iterations. Defaults to 500. Using default settings may not be a good idea.

c1 the weight towards the individual's best solution. Typically between 0 and 2; defaults to 1. Using default settings may not be a good idea. In some cases, even negative values work well: the solution is then driven off its past best position. For 'simple' problems, setting c1 to zero may work well: the population moves then towards the best overall solution.

c2 the weight towards the populations's best solution. Typically between 0 and 2; defaults to 1. Using default settings may not be a good idea. In some cases, even negative values work well: the solution is then driven off the population's past best position.

iner the inertia weight (a scalar), which reduces velocity. Typically between 0 and 1. Default is 0.9.

initV the standard deviation of the initial velocities. Defaults to 1.

maxV the maximum (absolute) velocity. Setting limits to velocity is sometimes called velocity clamping. Velocity is the change in a given solution in a given iteration. A maximum velocity can be set so to prevent unreasonable velocities ('overshooting'): for instance, if a decision variable may lie between 0 and 1, then an absolute velocity much greater than 1 makes rarely sense.

min, max vectors of minimum and maximum parameter values. The vectors min and max are used to determine the dimension of the problem and to randomly initialise the population. Per default, they are no constraints: a solution may well be outside these limits. Only if algo$minmaxConstr is TRUE will the algorithm repair solutions outside the min and max range.

minmaxConstr if TRUE, algo$min and algo$max are considered constraints. Default is FALSE.

pen a penalty function. Default is NULL (no penalty).

repair a repair function. Default is NULL (no repairing).

changeV a function to change velocity. Default is NULL (no change). This function is called before the velocity is added to the current solutions; it can be used to impose restrictions like changing only a number of decision variables.

initP optional: the initial population. A matrix of size length(algo$min) times algo$nP, or a function that creates such a matrix. If a function, it should take no arguments.

loopOF logical. Should the OF be evaluated through a loop? Defaults to TRUE.

loopPen logical. Should the penalty function (if specified) be evaluated through a loop? Defaults to TRUE.

loopRepair logical. Should the repair function (if specified) be evaluated through a loop? Defaults to TRUE.

loopChangeV logical. Should the changeV function (if specified) be evaluated through a loop? Defaults to TRUE.

printDetail If TRUE (the default), information is printed. If an integer i greater then one, information is printed at very ith iteration.

printBar If TRUE (the default), a txtProgressBar (from package **utils**) is printed).

storeF If TRUE (the default), the objective function values for every solution in every generation are stored and returned as matrix Fmat.

storeSolutions default is FALSE. If TRUE, the solutions (ie, decision variables) in every generation are stored as lists P and Pbest, both stored in the list xlist which the function returns. To check, for instance, the solutions at the end of the ith iteration, retrieve xlist[[c(1L, i)]]; the best solutions at the end of this iteration are in xlist[[c(2L, i)]]. P[[i]] and Pbest[[i]] will be matrices of size length(algo$min) times algo$nP.

Value

Returns a list:

`xbest`	the solution
`OFvalue`	objective function value of best solution
`popF`	a vector: the objective function values in the final population
`Fmat`	if `algo$storeF` is `TRUE`, a matrix of size `algo$nG` times `algo$nP`. Each column contains the best objective function value found by the particular solution.
`xlist`	if `algo$storeSolutions` is `TRUE`, a list that contains two lists `P` and `Pbest` of matrices, and a matrix `initP` (the initial solution); else `NA`.
`initial.state`	the value of `.Random.seed` when `PSopt` was called.

Author(s)

Enrico Schumann

References

Eberhart, R.C. and Kennedy, J. (1995) A New Optimizer using Particle Swarm theory. *Proceedings of the Sixth International Symposium on Micromachine and Human Science*, pp. 39–43.

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2017) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

See Also

`DEopt`

Examples

```
## Least Median of Squares (LMS) estimation
genData <- function(nP, nO, ol, dy) {
    ## create dataset as in Salibian-Barrera & Yohai 2006
    ## nP = regressors, nO  = number of obs
    ## ol = number of outliers, dy  = outlier size
    mRN <- function(m, n) array(rnorm(m * n), dim = c(m, n))
    y <- mRN(nO, 1)
    X <- cbind(as.matrix(numeric(nO) + 1), mRN(nO, nP - 1L))
    zz <- sample(nO)
    z  <- cbind(1, 100, array(0, dim = c(1L, nP - 2L)))
    for (i in seq_len(ol)) {
        X[zz[i], ] <- z
        y[zz[i]] <- dy
    }
    list(X = X, y = y)
}
```

```
OF <- function(param, data) {
    X <- data$X
    y <- data$y
    aux <- as.vector(y) - X %*% param
    ## as.vector(y) for recycling (param is a matrix)
    aux <- aux * aux
    aux <- apply(aux, 2, sort, partial = data$h)
    aux[h, ]
}

nP <- 2L; nO <- 100L; ol <- 10L; dy <- 150
aux <- genData(nP,nO,ol,dy); X <- aux$X; y <- aux$y

h <- (nO + nP + 1L) %/% 2
data <- list(y = y,X = X, h = h)

algo <- list(min = rep(-10, nP), max = rep( 10, nP),
    c1 = 1.0, c2 = 2.0,
    iner = 0.7, initV = 1, maxV = 3,
    nP = 100L, nG = 300L, loopOF = FALSE)

system.time(sol <- PSopt(OF = OF, algo = algo, data = data))
if (require("MASS", quietly = TRUE)) {
    ## for nsamp = "best", in this case, complete enumeration
    ## will be tried. See ?lqs
    system.time(test1 <- lqs(data$y ~ data$X[ ,-1L],
            adjust = TRUE,
            nsamp = "best",
            method = "lqs",
            quantile = data$h))
}
## check
x1 <- sort((y - X %*% as.matrix(sol$xbest))^2)[h]
cat("Particle Swarm\n",x1,"\n\n")

if (require("MASS", quietly = TRUE)) {
    x2 <- sort((y - X %*% as.matrix(coef(test1)))^2)[h]
    cat("lqs\n",x2,"\n\n")
}
```

---

putCallParity *Put-Call Parity*

Description

Put–call parity

Usage

```
putCallParity(what, call, put, S, X, tau, r, q = 0, tauD = 0, D = 0)
```

Arguments

what	character: what to compute. Currently only call or put are supported.
call	call price
put	put price
S	underlier
X	strike
tau	time to expiry
r	interest rate
q	dividend rate
tauD	numeric vector: time to dividend
D	numeric vector: dividends

Details

Put–call parity only works for European options. The function is vectorised (like vanillaOptionEuropean), except for dividends.

Value

Numeric vector.

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

Examples

```
S <- 100; X <- 100; tau <- 1; r <- 0.02; q <- 0.0;
vol <- 0.3; D <- 20; tauD <- 0.5
call <- vanillaOptionEuropean(S, X, tau, r, q, vol^2,
                              tauD = tauD, D = D, type = "call")$value
put  <- vanillaOptionEuropean(S, X, tau, r, q, vol^2,
                              tauD = tauD, D = D, type = "put")$value

## recover the call from the put (et vice versa)
all.equal(call, putCallParity("call", put = put,  S=S, X=X, tau=tau,
                              r=r, q=q, tauD=tauD, D=D))
all.equal(put, putCallParity("put",  call = call, S=S, X=X, tau=tau,
                             r=r, q=q, tauD=tauD, D=D))
```

```
## Black--Scholes--Merton with with 'callCF'
S <- 100; X <- 90; tau <- 1; r <- 0.02; q <- 0.08
v <- 0.2^2  ## variance, not volatility

(ccf <- callCF(cf = cfBSM, S = S, X = X, tau = tau, r = r, q = q,
                v = v, implVol = TRUE))
all.equal(ccf$value,
          vanillaOptionEuropean(S, X, tau, r, q, v, type = "call")$value)
all.equal(
  putCallParity("put", call=ccf$value, S=S, X=X, tau=tau, r=r, q=q),
  vanillaOptionEuropean(S, X, tau, r, q, v, type = "put")$value)
```

---

qTable *Prepare LaTeX Table with Quartile Plots*

Description

The function returns the skeleton of a LaTeX tabular that contains the median, minimum and maximum of the columns of a matrix `X`. For each column, a quartile plot is added.

Usage

```
qTable(X, xmin = NULL, xmax = NULL, labels = NULL, at = NULL,
       unitlength = "5cm", linethickness = NULL,
       cnames = colnames(X), circlesize = 0.01,
       xoffset = 0, yoffset = 0, dec = 2, filename = NULL,
       funs = list(median = median, min = min, max = max),
       tabular.format, skip = TRUE)
```

Arguments

`X`	a numeric matrix (or an object that can be coerced to a numeric matrix with `as.matrix`)
`xmin`	optional: the minimum for the x-axis. See Details.
`xmax`	optional: the maximum for the x-axis. See Details.
`labels`	optional: labels for the x-axis.
`at`	optional: where to put labels.
`unitlength`	the unitlength for LaTeX's `picture` environment. See Details.
`linethickness`	the linethickness for LaTeX's `picture` environment. See Details.
`cnames`	the column names of `X`
`circlesize`	the size of the circle in LaTeX's `picture` environment
`xoffset`	defaults to 0. See Details.
`yoffset`	defaults to 0. See Details.
`dec`	the number of decimals

filename	if provided, output is cat into a file
funs	A list of functions; the functions should be named. Default is list(median = median, min = min, max = max)
tabular.format	optional: character string like "rrrrr" that defines the format of the tabular.
skip	Adds a newline at the end of the tabular. Default is TRUE. (The behaviour prior to package version 0.27-0 corresponded to FALSE.)

Details

The function creates a one-column character matrix that can be put into a LaTeX file (the matrix holds a tabular). It relies on LaTeX's picture environment and should work for LaTeX and pdfLaTeX. Note that the tabular needs generally be refined, depending on the settings and the data.

The tabular has one row for every column of X (and header and footer rows). A given row contains (per default) the median, the minimum and the maximum of the column; it also includes a picture environment the shows a quartile plot of the distribution of the elements in that column. Other functions can be specified via argument funs.

A number of parameters can be passed to LaTeX's picture environment: unitlength, xoffset, yoffset, linethickness. Sizes and lengths are functions of unitlength (linethickness is an exception; and while circlesize is a multiple of unitlength, it will not translate into an actual diameter of more than 14mm).

The whole tabular environment is put into curly brackets so that the settings do not change settings elsewhere in the LaTeX document.

If xmin, xmax, labels and at are not specified, they are computed through a call to pretty from the **base** package. If limits are specified, then both xmin and xmax must be set; if labels are used, then both labels and at must be specified.

To use the function in a vignette, use cat(tTable(X)) (and results=tex in the code chunk options). The vignette qTableEx shows some examples.

Value

A matrix of mode character. If filename is specified then qTable will have the side effect of writing a textfile with a LaTeX tabular.

Note

qTable returns a raw draft of a table for LaTeX. Please, spend some time on making it pretty.

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Tufte, E. (2001) *The Visual Display of Quantitative Information*. 2nd edition, Graphics Press.

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

Examples

```
x <- rnorm(100, mean = 0, sd = 2)
y <- rnorm(100, mean = 1, sd = 2)
z <- rnorm(100, mean = 1, sd = 0.5)
X <- cbind(x, y, z)
res <- qTable(X)
print(res)
cat(res)

## Not run:
## show vignette with examples
qt <- vignette("qTableEx", package = "NMOF")
print(qt)
edit(qt)


## create a simple LaTeX file 'test.tex':
## ---
##  \documentclass{article}
##  \begin{document}
##    \input{res.tex}
##  \end{document}
## ---

res <- qTable(X, filename = "res.tex", yoffset = -0.025, unitlength = "5cm",
              circlesize = 0.0125, xmin = -10, xmax = 10, dec = 2)
## End(Not run)
```

`repairMatrix` *Repair an Indefinite Correlation Matrix*

Description

The function 'repairs' an indefinite correlation matrix by replacing its negative eigenvalues by zero.

Usage

```
repairMatrix(C, eps = 0)
```

Arguments

`C`	a correlation matrix
`eps`	a small number

Details

The function 'repairs' a correlation matrix: it replaces negative eigenvalues with eps and rescales the matrix such that all elements on the main diagonal become unity again.

Value

Returns a numeric matrix.

Note

This function may help to cure a numerical problem, but it will rarely help to cure an empirical problem. (Garbage in, garbage out.)

See also the function nearPD in the **Matrix** package.

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Rebonato, R. and Jaeckel, P. (1999) The most general methodology to create a valid correlation matrix for risk management and option pricing purposes.

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

Examples

```
## example: build a portfolio of three assets
C <- c(1,.9,.9,.9,1,.2,.9,.2,1)
dim(C) <- c(3L, 3L)
eigen(C, only.values = TRUE)

vols <- c(.3, .3, .3)      ## volatilities
S <- C * outer(vols,vols)  ## covariance matrix
w <- c(-1, 1, 1)           ## a portfolio
w %*% S %*% w          ## variance of portfolio is negative!
sqrt(as.complex(w %*% S %*% w))

S <- repairMatrix(C) * outer(vols,vols)
w %*% S %*% w          ## more reasonable
sqrt(w %*% S %*% w)
```

resampleC *Resample with Specified Rank Correlation*

Description

Resample with replacement from a number of vectors; the sample will have a specified rank correlation.

Usage

```
resampleC(..., size, cormat)
```

Arguments

...	numeric vectors; they need not have the same length.
size	an integer: the number of samples to draw
cormat	the rank correlation matrix

Details

See Gilli, Maringer and Schumann (2011), Section 7.1.2. The function samples with replacement from the vectors passed through .... The resulting samples will have an (approximate) rank correlation as specified in cormat.

The function uses the eigenvalue decomposition to generate the correlation; it will not break down in case of a semidefinite matrix. If an eigenvalue of cormat is smaller than zero, a warning is issued (but the function proceeds).

Value

a numeric matrix with size rows. The columns contain the samples; hence, there will be as many columns as vectors passed through ....

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

See Also

repairMatrix

Examples

```
## a sample
v1 <- rnorm(20)
v2 <- runif(50)
v3 <- rbinom(100, size = 50, prob = 0.4)

## a correlation matrix
cormat <- array(0.5, dim = c(3, 3))
diag(cormat) <- 1

cor(resampleC(a = v1, b = v2, v3, size = 100, cormat = cormat),
    method = "spearman")
```

`restartOpt` *Restart an Optimisation Algorithm*

Description

The function provides a simple wrapper for the optimisation algorithms in the package.

Usage

```
restartOpt(fun, n, OF, algo, ...,
           method = c("loop", "multicore", "snow"),
           mc.control = list(), cl = NULL,
           best.only = FALSE)
```

Arguments

`fun`	the optimisation function: `DEopt`, `GAopt`, `LSopt`, `TAopt` or `PSopt`
`n`	the number of restarts
`OF`	the objective function
`algo`	the list `algo` that is passed to the particular optimisation function
`...`	additional data that is passed to the particular optimisation function
`method`	can be `loop` (the default), `multicore` or `snow`. See Details.
`mc.control`	a list containing settings that will be passed to `mclapply` if `method` is `multicore`. Must be a list of named elements. See the documentation of `mclapply`.
`cl`	default is `NULL`. If method `snow` is used, this must be a cluster object or an integer (the number of cores).
`best.only`	if `TRUE`, only the best run is reported. Default is `FALSE`.

Details

The function returns a list of lists. If a specific starting solution is passed, all runs will start from this solution. If this is not desired, initial solutions can be created randomly. This is done per default in DEopt, GAopt and PSopt, but LSopt and TAopt require to specify a starting solution.

In case of LSopt and TAopt, the passed initial solution algo$x0 is checked with is.function: if TRUE, the function is evaluated in each single run. For DEopt, GAopt and PSopt, the initial solution (which also can be a function) is specified with algo$initP.

The argument method determines how fun is evaluated. Default is loop. If method is "multicore", function mclapply from package **parallel** is used. Further settings for mclapply can be passed through the list mc.control. If multicore is chosen but the functionality is not available, then method will be set to loop and a warning is issued. If method == "snow", function clusterApply from package **parallel** is used. In this case, the argument cl must either be a cluster object (see the documentation of clusterApply) or an integer. If an integer, a cluster will be set up via makeCluster(c(rep("localhost", cl)), type = "SOCK"), and stopCluster is called when the function is exited. If snow is chosen but **parallel** is not available or cl is not specified, then method will be set to loop and a warning is issued. In case that cl is an cluster object, stopCluster will not be called automatically.

Value

If best.only is FALSE (the default), the function returns a list of n lists. Each of the n lists stores the output of one of the runs.

If best.only is TRUE, only the best restart is reported. The returned list has the structure specific to the used method.

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

See Also

DEopt, GAopt, LSopt, PSopt, TAopt

Examples

```
## see example(DEopt)
algo <- list(nP = 50L,
             F = 0.5,
            CR = 0.9,
           min = c(-10, -10),
           max = c( 10,  10),
```

```
    printDetail = FALSE,
       printBar = FALSE)

## choose a larger 'n' when you can afford it
algo$nG <- 100L
res100 <- restartOpt(DEopt, n = 5L, OF = tfTrefethen, algo = algo)
res100F <- sapply(res100, `[[`, "OFvalue")

algo$nG <- 200L
res200 <- restartOpt(DEopt, n = 5L, OF = tfTrefethen, algo = algo)
res200F <- sapply(res200, `[[`, "OFvalue")

xx <- pretty(c(res100F, res200F, -3.31))
plot(ecdf(res100F), main = "optimum is -3.306",
     xlim = c(xx[1L], tail(xx, 1L)))
abline(v = -3.3069, col = "red")  ## optimum
lines(ecdf(res200F), col = "blue")
legend(x = "right", box.lty = 0, , lty = 1,
      legend = c("optimum", "100 generations", "200 generations"),
      pch = c(NA, 19, 19), col = c("red", "black", "blue"))

## a 'best-of-N' strategy: given a sample x of objective
## function values, compute the probability that, after N draws,
## we have at least one realisation not worse than X
x <- c(0.1,.3,.5,.5,.6)
bestofN <- function(x, N) {
    nx <- length(x)
    function(X)
        1 - (sum(x > X)/nx)^N
}
bestof2 <- bestofN(x, 2)
bestof5 <- bestofN(x, 5)
bestof2(0.15)
bestof5(0.15)


## Not run:
## with R >= 2.13.0 and the compiler package
algo$nG <- 100L
system.time(res100 <- restartOpt(DEopt, n = 10L, OF = tfTrefethen, algo = algo))

require("compiler")
enableJIT(3)
system.time(res100 <- restartOpt(DEopt, n = 10L, OF = tfTrefethen, algo = algo))

## End(Not run)
```

SA.info *Simulated-Annealing Information*

Description

The function can be called from the objective and neighbourhood function during a run of SAopt; it provides information such as the current iteration, the current solution, etc.

Usage

```
SA.info(n = 0L)
```

Arguments

n	generational offset; see Details.

Details

This function is still experimental.

The function can be called in the neighbourhood function or the objective function during a run of SAopt. It evaluates to a list with information about the state of the optimisation run, such as the current iteration or the currently best solution.

SA.info relies on parent.frame to retrieve its information. If the function is called within another function within the neighbourhood or objective function, the argument n needs to be increased.

Value

A list

calibration	logical: whether the algorithm is calibrating the acceptance probability
iteration	current iteration
step	current step for the given temperature level
temperature	current temperature (the number, not the value)
xbest	the best solution found so far

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2017) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

See Also

SAopt, TA.info

Examples

```
### MINIMAL EXAMPLE for SAopt

## the objective function evaluates to a constant
fun <- function(x)
    0

## the neighbourhood function does not even change
## the solution; it only reports information
nb <- function(x) {
    info <- SA.info()
    cat("current step ",        info$step,
        "| current iteration ", info$iteration, "\n")
    x
}

## run SA
algo <- list(nS = 5, nT = 2, nD = 10,
             initT = 1,
             x0 = rep(0, 5),
             neighbour = nb,
             printBar = FALSE)
ignore <- SAopt(fun, algo)
```

SAopt *Optimisation with Simulated Annealing*

Description

The function implements a Simulated-Annealing algorithm.

Usage

```
SAopt(OF, algo = list(), ...)
```

Arguments

OF	The objective function, to be minimised. Its first argument needs to be a solution x; it will be called as OF(x, ...).
algo	A list of settings for the algorithm. See Details.
...	other variables passed to OF and algo$neighbour. See Details.

Details

Simulated Annealing (SA) changes an initial solution iteratively; the algorithm stops after a fixed number of iterations. Conceptually, SA consists of a loop than runs for a number of iterations. In each iteration, a current solution xc is changed through a function algo$neighbour. If this new (or neighbour) solution xn is not worse than xc, ie, if OF(xn,...) <= OF(xc,...), then xn replaces

xc. If xn is worse, it still replaces xc, but only with a certain probability. This probability is a function of the degree of the deterioration (the greater, the less likely the new solution is accepted) and the current iteration (the longer the algorithm has already run, the less likely the new solution is accepted).

The list algo contains the following items.

nS The number of steps per temperature. The default is 1000; but this setting depends very much on the problem.

nT The number of temperatures. Default is 10.

nD The number of random steps to calibrate the temperature. Defaults to 2000.

initT Initial temperature. Defaults to NULL, in which case it is automatically chosen so that initProb is achieved.

finalT Final temperature. Defaults to 0.

alpha The cooling constant. The current temperature is multiplied by this value. Default is 0.9.

mStep Step multiplier. The default is 1, which implies constant number of steps per temperature. If greater than 1, the step number nS is increased to m*nS (and rounded).

x0 The initial solution. If this is a function, it will be called once without arguments to compute an initial solution, ie, x0 <- algo$x0(). This can be useful when the routine is called in a loop of restarts, and each restart is to have its own starting value.

neighbour The neighbourhood function, called as neighbour(x, ...). Its first argument must be a solution x; it must return a changed solution.

printDetail If TRUE (the default), information is printed. If an integer i greater then one, information is printed at very ith iteration.

printBar If TRUE (default is FALSE), a txtProgressBar (from package **utils**) is printed. The progress bar is not shown if printDetail is an integer greater than 1.

storeF if TRUE (the default), the objective function values for every solution in every generation are stored and returned as matrix Fmat.

storeSolutions Default is FALSE. If TRUE, the solutions (ie, decision variables) in every generation are stored and returned in list xlist (see Value section below). To check, for instance, the current solution at the end of the ith generation, retrieve xlist[[c(2L, i)]].

classify Logical; default is FALSE. If TRUE, the result will have a class attribute SAopt attached.

OF.target Numeric; when specified, the algorithm will stop when an objective-function value as low as OF.target (or lower) is achieved. This is useful when an optimal objective-function value is known: the algorithm will then stop and not waste time searching for a better solution.

At the minimum, algo needs to contain an initial solution x0 and a neighbour function.

The total number of iterations equals algo$nT times algo$nS (plus possibly algo$nD).

Value

SAopt returns a list with five components:

xbest	the solution
OFvalue	objective function value of the solution, ie, OF(xbest, ...)

Fmat	if algo$storeF is TRUE, a matrix with one row for each iteration (excluding the initial algo$nD steps) and two columns. The first column contains the objective function values of the neighbour solution at a given iteration; the second column contains the value of the current solution. Since SA can walk away from locally-optimal solutions, the best solution can be monitored through cummin(Fmat[,2L]).
xlist	if algo$storeSolutions is TRUE, a list; else NA. Contains the neighbour solutions at a given iteration (xn) and the current solutions (xc). Example: Fmat[i, 2L] is the objective function value associated with xlist[[c(2L, i)]].
initial.state	the value of .Random.seed when the function was called.

If algo$classify was set to TRUE, the resulting list will have a class attribute TAopt.

Note

If the ... argument is used, then all the objects passed with ... need to go into the objective function and the neighbourhood function. It is recommended to collect all information in a list myList and then write OF and neighbour so that they are called as OF(x, myList) and neighbour(x, myList). Note that x need not be a vector but can be any data structure (eg, a matrix or a list).

Using an initial and final temperature of zero means that SA will be equivalent to a Local Search. The function LSopt may be preferred then because of smaller overhead.

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Kirkpatrick, S., Gelatt, C.D. and Vecchi, M.P. (1983). Optimization with Simulated Annealing. Science. **220** (4598), 671–680.

Schumann, E. (2017) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

See Also

LSopt, TAopt, restartOpt

Examples

```
## Aim: given a matrix x with n rows and 2 columns,
##      divide the rows of x into two subsets such that
##      in one subset the columns are highly correlated,
##      and in the other lowly (negatively) correlated.
##      constraint: a single subset should have at least 40 rows

## create data with specified correlation
```

```
n <- 100L
rho <- 0.7
C <- matrix(rho, 2L, 2L); diag(C) <- 1
x <- matrix(rnorm(n * 2L), n, 2L) %*% chol(C)

## collect data
data <- list(x = x, n = n, nmin = 40L)

## a random initial solution
x0 <- runif(n) > 0.5

## a neighbourhood function
neighbour <- function(xc, data) {
    xn <- xc
    p <- sample.int(data$n, size = 1L)
    xn[p] <- abs(xn[p] - 1L)
    # reject infeasible solution
    c1 <- sum(xn) >= data$nmin
    c2 <- sum(xn) <= (data$n - data$nmin)
    if (c1 && c2) res <- xn else res <- xc
    as.logical(res)
}

## check (should be 1 FALSE and n-1 TRUE)
x0 == neighbour(x0, data)

## objective function
OF <- function(xc, data)
    -abs(cor(data$x[xc, ])[1L, 2L] - cor(data$x[!xc, ])[1L, 2L])

## check
OF(x0, data)
## check
OF(neighbour(x0, data), data)

## plot data
par(mfrow = c(1,3), bty = "n")
plot(data$x,
     xlim = c(-3,3), ylim = c(-3,3),
     main = "all data", col = "darkgreen")

## *Local Search*
algo <- list(nS = 3000L,
             neighbour = neighbour,
             x0 = x0,
             printBar = FALSE)
sol1 <- LSopt(OF, algo = algo, data=data)
sol1$OFvalue

## *Simulated Annealing*
algo$nT <- 10L
algo$nS <- ceiling(algo$nS/algo$nT)
sol <- SAopt(OF, algo = algo, data = data)
```

```
sol$OFvalue

c1 <- cor(data$x[ sol$xbest, ])[1L, 2L]
c2 <- cor(data$x[!sol$xbest, ])[1L, 2L]

lines(data$x[ sol$xbest, ], type = "p", col = "blue")

plot(data$x[ sol$xbest, ], col = "blue",
     xlim = c(-3, 3), ylim = c(-3, 3),
     main = paste("subset 1, corr.", format(c1, digits = 3)))

plot(data$x[!sol$xbest, ], col = "darkgreen",
     xlim = c(-3,3), ylim = c(-3,3),
     main = paste("subset 2, corr.", format(c2, digits = 3)))

## compare LS/SA
par(mfrow = c(1, 1), bty = "n")
plot(sol1$Fmat[ , 2L],type = "l", ylim=c(-1.5, 0.5),
     ylab = "OF", xlab = "Iterations")
lines(sol$Fmat[ , 2L],type = "l", col = "blue")
legend(x = "topright", legend = c("LS", "SA"),
       lty = 1, lwd = 2, col = c("black", "blue"))
```

`showExample` *Display examples*

Description

Display the code examples from 'Numerical Methods and Optimization and Finance'.

Usage

```
showExample(file = "", chapter = NULL, showfile = TRUE,
            includepaths = FALSE, ...)
showChapterNames()
```

Arguments

`file`	a character vector of length one. See Details.
`chapter`	optional: a character vector of length one, giving the chapter name (see Details), or an integer, indicating a chapter number. Default is `NULL`: look in all chapters.
`showfile`	Should the file be displayed with `file.show`? Defaults to `TRUE`. A file will be displayed if only a single file is identified by `file` and `chapter`.
`includepaths`	Should the file paths be displayed? Defaults to `FALSE`.
`...`	Arguments passed to `grepl`; see Details.

Details

showExample matches the specified file argument against the available file names via grepl(file, filenames, ...). If chapter is specified, a second match is performed, grepl(chapter, chapternames, ...). The chapternames are those in the book (eg, 'Modeling dependencies'). The selected files are then those for which file name and chapter name could be matched.

Value

showExample returns a data.frame of at least two character vectors, Chapter and File. If includepaths is TRUE, Paths are included. If no file is found, the data.frame has zero rows. If a single file is identified and showfile is TRUE, the function has the side effect of displaying that file.

showChapterNames returns a character vector of length 15: the names of the book's chapters.

Note

The files can also be downloaded from http://nmof.net .

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

Examples

```
showExample("equations.R")

showExample("example", chapter = "portfolio")
showExample("example", chapter = "Portfolio")
showExample("example", chapter = "portfolio", ignore.case = TRUE)
showExample("example", chapter = 13)
showExample("example", chapter = showChapterNames()[13L])
```

TA.info *Threshold-Accepting Information*

Description

The function can be called from the objective and neighbourhood function during a run of TAopt; it provides information such as the current iteration, the current solution, etc.

Usage

```
TA.info(n = 0L)
```

Arguments

n	generational offset; see Details.

Details

This function is still experimental.

The function can be called in the neighbourhood function or the objective function during a run of TAopt. It evaluates to a list with the state of the optimisation run, such as the current iteration.

TA.info relies on parent.frame to retrieve its information. If the function is called within another function in the neighbourhood or objective function, the argument n needs to be increased.

Value

A list

OF.sampling	logical: if TRUE, is the algorithm sampling the objective function to compute thresholds; otherwise (i.e. during the actual optimisation) FALSE
iteration	current iteration
step	current step (i.e. for a given threshold)
threshold	current threshold (the number, not the value)
xbest	the best solution found so far

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

See Also

TAopt

Examples

```
### MINIMAL EXAMPLE for TAopt

## objective function evaluates to a constant
fun <- function(x)
    0

## neighbourhood function does not even change the solution,
## but it reports information
nb <- function(x) {
    tmp <- TA.info()
    cat("current threshold ",   tmp$threshold,
        "| current step ",      tmp$step,
        "| current iteration ", tmp$iteration, "\n")
    x
}

## run TA
algo <- list(nS = 5,
             nT = 2,
             nD = 3,
             x0 = rep(0, 5),
             neighbour = nb,
             printBar = FALSE,
             printDetail = FALSE)
ignore <- TAopt(fun, algo)

## printed output:
##    current threshold  NA | current step  1 | current iteration  NA
##    current threshold  NA | current step  2 | current iteration  NA
##    current threshold  NA | current step  3 | current iteration  NA
##    current threshold   1 | current step  1 | current iteration  1
##    current threshold   1 | current step  2 | current iteration  2
##    current threshold   1 | current step  3 | current iteration  3
##    current threshold   1 | current step  4 | current iteration  4
##    current threshold   1 | current step  5 | current iteration  5
##    current threshold   2 | current step  1 | current iteration  6
##    current threshold   2 | current step  2 | current iteration  7
##    current threshold   2 | current step  3 | current iteration  8
##    current threshold   2 | current step  4 | current iteration  9
##    current threshold   2 | current step  5 | current iteration  10
```

TAopt *Optimisation with Threshold Accepting*

Description

The function implements the Threshold Accepting algorithm.

Usage

```
TAopt(OF, algo = list(), ...)
```

Arguments

OF	The objective function, to be minimised. Its first argument needs to be a solution x; it will be called as OF(x, ...).
algo	A list of settings for the algorithm. See Details.
...	other variables passed to OF and algo$neighbour. See Details.

Details

Threshold Accepting (TA) changes an initial solution iteratively; the algorithm stops after a fixed number of iterations. Conceptually, TA consists of a loop than runs for a number of iterations. In each iteration, a current solution xc is changed through a function algo$neighbour. If this new (or neighbour) solution xn is not worse than xc, ie, if OF(xn,...) <= OF(xc,...), then xn replaces xc. If xn is worse, it still replaces xc as long as the difference in 'quality' between the two solutions is less than a threshold tau; more precisely, as long as OF(xn,...) - tau <= OF(xc,...). Thus, we also accept a new solution that is worse than its predecessor; just not too much worse. The threshold is typically decreased over the course of the optimisation. For zero thresholds TA becomes a stochastic local search.

The thresholds can be passed through the list algo (see below). Otherwise, they are automatically computed through the procedure described in Gilli et al. (2006). When the thresholds are created automatically, the final threshold is always zero.

The list algo contains the following items.

nS The number of steps per threshold. The default is 1000; but this setting depends very much on the problem.

nT The number of thresholds. Default is 10; ignored if algo$vT is specified.

nD The number of random steps to compute the threshold sequence. Defaults to 2000. Only used if algo$vT is NULL.

q The highest quantile for the threshold sequence. Defaults to 0.5. Only used if algo$vT is NULL. If q is zero, TAopt will run with algo$nT zero-thresholds (ie, like a Local Search).

x0 The initial solution. If this is a function, it will be called once without arguments to compute an initial solution, ie, x0 <- algo$x0(). This can be useful when the routine is called in a loop of restarts, and each restart is to have its own starting value.

vT The thresholds. A numeric vector. If NULL (the default), TAopt will compute algo$nT thresholds. Passing threshold can be useful when similar problems are handled. Then the time to sample the objective function to compute the thresholds can be saved (ie, we save algo$nD function evaluations). If the thresholds are computed and algo$printDetail is TRUE, the time required to evaluate the objective function will be measured and an estimate for the remaining computing time is issued. This estimate is often very crude.

neighbour The neighbourhood function, called as neighbour(x, ...). Its first argument must be a solution x; it must return a changed solution.

printDetail If TRUE (the default), information is printed. If an integer i greater then one, information is printed at very ith iteration.

`printBar` If `TRUE` (default is `FALSE`), a `txtProgressBar` (from package **utils**) is printed. The progress bar is not shown if `printDetail` is an integer greater than 1.

`scale` The thresholds are multiplied by `scale`. Default is 1.

`stepUp` Defaults to `0`. If an integer greater than zero, then the thresholds are recycled, ie, `vT` is replaced by `rep(vT, algo$stepUp + 1)` (and the number of thresholds will be increased by `algo$nT` times `algo$stepUp`). This option works for supplied as well as computed thresholds. Practically, this will have the same effect as restarting from a returned solution. (In Simulated Annealing, this strategy goes by the name of 'reheating'.)

`storeF` if `TRUE` (the default), the objective function values for every solution in every generation are stored and returned as matrix `Fmat`.

`storeSolutions` Default is `FALSE`. If `TRUE`, the solutions (ie, decision variables) in every generation are stored and returned in list `xlist` (see Value section below). To check, for instance, the current solution at the end of the `i`th generation, retrieve `xlist[[c(2L, i)]]`.

`classify` Logical; default is `FALSE`. If `TRUE`, the result will have a class attribute `TAopt` attached. This feature is **experimental**: the supported methods (plot, summary) may change without warning.

`OF.target` Numeric; when specified, the algorithm will stop when an objective-function value as low as `OF.target` (or lower) is achieved. This is useful when an optimal objective-function value is known: the algorithm will then stop and not waste time searching for a better solution.

At the minimum, `algo` needs to contain an initial solution `x0` and a `neighbour` function.

The total number of iterations equals `algo$nT` times `(algo$stepUp + 1)` times `algo$nS` (plus possibly `algo$nD`).

Value

`TAopt` returns a list with four components:

`xbest`	the solution
`OFvalue`	objective function value of the solution, ie, `OF(xbest, ...)`
`Fmat`	if `algo$storeF` is `TRUE`, a matrix with one row for each iteration (excluding the initial `algo$nD` steps) and two columns. The first column contains the objective function values of the neighbour solution at a given iteration; the second column contains the value of the current solution. Since TA can walk away from locally-optimal solutions, the best solution can be monitored through `cummin(Fmat[ ,2L])`.
`xlist`	if `algo$storeSolutions` is `TRUE`, a list; else `NA`. Contains the neighbour solutions at a given iteration (`xn`) and the current solutions (`xc`). Example: `Fmat[i, 2L]` is the objective function value associated with `xlist[[c(2L, i)]]`.
`initial.state`	the value of `.Random.seed` when the function was called.

If `algo$classify` was set to `TRUE`, the resulting list will have a class attribute `TAopt`.

Note

If the ... argument is used, then all the objects passed with ... need to go into the objective function and the neighbourhood function. It is recommended to collect all information in a list myList and then write OF and neighbour so that they are called as OF(x, myList) and neighbour(x, myList). Note that x need not be a vector but can be any data structure (eg, a matrix or a list).

Using thresholds of size 0 makes TA run as a Local Search. The function LSopt may be preferred then because of smaller overhead.

Author(s)

Enrico Schumann

References

Dueck, G. and Scheuer, T. (1990) Threshold Accepting. A General Purpose Optimization Algorithm Superior to Simulated Annealing. *Journal of Computational Physics*. **90** (1), 161–175.

Dueck, G. and Winker, P. (1992) New Concepts and Algorithms for Portfolio Choice. *Applied Stochastic Models and Data Analysis*. **8** (3), 159–178.

Gilli, M., Kellezi, E. and Hysi, H. (2006) A Data-Driven Optimization Heuristic for Downside Risk Minimization. *Journal of Risk*. **8** (3), 1–18.

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Moscato, P. and Fontanari, J.F. (1990). Stochastic Versus Deterministic Update in Simulated Annealing. Physics Letters A. **146** (4), 204–208.

Schumann, E. (2012) Remarks on 'A comparison of some heuristic optimization methods'. http://enricoschumann.net/R/remarks.htm

Schumann, E. (2017) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

Winker, P. (2001). *Optimization Heuristics in Econometrics: Applications of Threshold Accepting*. Wiley.

See Also

LSopt, restartOpt

Examples

```
## Aim: given a matrix x with n rows and 2 columns,
##      divide the rows of x into two subsets such that
##      in one subset the columns are highly correlated,
##      and in the other lowly (negatively) correlated.
##      constraint: a single subset should have at least 40 rows

## create data with specified correlation
n <- 100L
rho <- 0.7
```

```
C <- matrix(rho, 2L, 2L); diag(C) <- 1
x <- matrix(rnorm(n * 2L), n, 2L) %*% chol(C)

## collect data
data <- list(x = x, n = n, nmin = 40L)

## a random initial solution
x0 <- runif(n) > 0.5

## a neighbourhood function
neighbour <- function(xc, data) {
    xn <- xc
    p <- sample.int(data$n, size = 1L)
    xn[p] <- abs(xn[p] - 1L)
    # reject infeasible solution
    c1 <- sum(xn) >= data$nmin
    c2 <- sum(xn) <= (data$n - data$nmin)
    if (c1 && c2) res <- xn else res <- xc
    as.logical(res)
}

## check (should be 1 FALSE and n-1 TRUE)
x0 == neighbour(x0, data)

## objective function
OF <- function(xc, data)
    -abs(cor(data$x[xc, ])[1L, 2L] - cor(data$x[!xc, ])[1L, 2L])

## check
OF(x0, data)
## check
OF(neighbour(x0, data), data)

## plot data
par(mfrow = c(1,3), bty = "n")
plot(data$x,
     xlim = c(-3,3), ylim = c(-3,3),
     main = "all data", col = "darkgreen")

## *Local Search*
algo <- list(nS = 3000L,
             neighbour = neighbour,
             x0 = x0,
             printBar = FALSE)
sol1 <- LSopt(OF, algo = algo, data=data)
sol1$OFvalue

## *Threshold Accepting*
algo$nT <- 10L
algo$nS <- ceiling(algo$nS/algo$nT)
sol <- TAopt(OF, algo = algo, data = data)
sol$OFvalue
```

```
c1 <- cor(data$x[ sol$xbest, ])[1L, 2L]
c2 <- cor(data$x[!sol$xbest, ])[1L, 2L]

lines(data$x[ sol$xbest, ], type = "p", col = "blue")

plot(data$x[ sol$xbest, ], col = "blue",
     xlim = c(-3,3), ylim = c(-3,3),
     main = paste("subset 1, corr.", format(c1, digits = 3)))

plot(data$x[!sol$xbest, ], col = "darkgreen",
     xlim = c(-3,3), ylim = c(-3,3),
     main = paste("subset 2, corr.", format(c2, digits = 3)))

## compare LS/TA
par(mfrow = c(1,1), bty = "n")
plot(sol1$Fmat[ ,2L],type="l", ylim=c(-1.5,0.5),
     ylab = "OF", xlab = "iterations")
lines(sol$Fmat[ ,2L],type = "l", col = "blue")
legend(x = "topright",legend = c("LS", "TA"),
       lty = 1, lwd = 2,col = c("black", "blue"))
```

`testFunctions` *Classical Test Functions for Unconstrained Optimisation*

Description

A number of functions that have been suggested in the literature as benchmarks for unconstrained optimisation.

Usage

```
tfAckley(x)
tfEggholder(x)
tfGriewank(x)
tfRastrigin(x)
tfRosenbrock(x)
tfSchwefel(x)
tfTrefethen(x)
```

Arguments

x	a numeric vector of arguments. See Details.

Details

All functions take as argument only one variable, a numeric vector `x` whose length determines the dimensionality of the problem.

The *Ackley* function is implemented as

$$\exp(1) + 20 - 20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right).$$

The minimum function value is zero; reached at $x = 0$.

The *Eggholder* takes a two-dimensional `x`, here written as x and y. It is defined as

$$-(y+47)\sin\left(\sqrt{|y + \frac{x}{2} + 47|}\right) - x\sin\left(\sqrt{|x - (y+47)|}\right).$$

The minimum function value is -959.6407; reached at `c(512, 404.2319)`.

The *Griewank* function is given by

$$1 + \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n}\cos\left(\frac{x_i}{\sqrt{i}}\right).$$

The function is minimised at $x = 0$; its minimum value is zero.

The *Rastrigin* function:

$$10n + \sum_{i=1}^{n}\left(x_i^2 - 10\cos(2\pi x_i)\right).$$

The minimum function value is zero; reached at $x = 0$.

The *Rosenbrock* (or banana) function:

$$\sum_{i=1}^{n-1}\left(100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right).$$

The minimum function value is zero; reached at $x = 1$.

The *Schwefel* function:

$$\sum_{i=1}^{n}\left(-x_i\sin\left(\sqrt{|x_i|}\right)\right).$$

The minimum function value (to about 8 digits) is $-418.9829n$; reached at $x = 420.9687$.

Trefethen's function takes a two-dimensional `x` (here written as x and y); it is defined as

$$\exp(\sin(50x)) + \sin(60e^y) + \sin(70\sin(x)) + \sin(\sin(80y)) - \sin(10(x+y)) + \frac{1}{4}(x^2 + y^2).$$

The minimum function value is -3.3069; reached at `c(-0.0244, 0.2106)`.

Value

The objective function evaluated at `x` (a numeric vector of length one).

Warning

These test functions represent *artificial* problems. It is practically not too helpful to fine-tune a method on such functions. (That would be like memorising all the answers to a particular multiple-choice test.) The functions' main purpose is checking the numerical implementation of algorithms.

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626 (Chapter 10)

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

See Also

DEopt, PSopt

Examples

```
## persp for two-dimensional x

## Ackley
n <- 100L; surf <- matrix(NA, n, n)
x1 <- seq(from = -2, to = 2, length.out = n)
for (i in 1:n)
    for (j in 1:n)
        surf[i, j] <- tfAckley(c(x1[i], x1[j]))
persp(x1, x1, -surf, phi = 30, theta = 30, expand = 0.5,
      col = "goldenrod1", shade = 0.2, ticktype = "detailed",
      xlab = "x1", ylab = "x2", zlab = "-f", main = "Ackley (-f)",
      border = NA)

## Trefethen
n <- 100L; surf <- matrix(NA, n, n)
x1 <- seq(from = -10, to = 10, length.out = n)
for (i in 1:n)
    for (j in 1:n)
        surf[i, j] <- tfTrefethen(c(x1[i], x1[j]))
persp(x1, x1, -surf, phi = 30, theta = 30, expand = 0.5,
      col = "goldenrod1", shade = 0.2, ticktype = "detailed",
      xlab = "x1", ylab = "x2", zlab = "-f", main = "Trefethen (-f)",
      border = NA)
```

---

vanillaBond *Pricing Plain-Vanilla Bonds*

Description

Calculate the theoretical price and yield-to-maturity of a list of cashflows.

Usage

```
vanillaBond(cf, times, df, yields)
ytm(cf, times, y0 = 0.05, tol = 1e-05, maxit = 1000L, offset = 0)

duration(cf, times, yield, modified = TRUE, raw = FALSE)
convexity(cf, times, yield, raw = FALSE)
```

Arguments

cf	Cashflows; a numeric vector or a matrix. If a matrix, cashflows should be arranged in rows; times-to-payment correspond to columns.
times	times-to-payment; a numeric vector
df	discount factors; a numeric vector
yields	optional (instead of discount factors); zero yields to compute discount factor; if of length one, a flat zero curve is assumed
yield	numeric vector of length one (both duration and convexity assume a flat yield curve)
y0	starting value
tol	tolerance
maxit	maximum number of iterations
offset	numeric: a 'base' rate over which to compute the yield to maturity. See Details and Examples.
modified	logical: return modified duration? (default TRUE)
raw	logical: default FALSE. Compute duration/convexity as derivative of cashflows' present value? Use this if you want to approximate the change in the bond price by a Taylor series (see Examples).

Details

vanillaBond computes the present value of a vector of cashflows; it may thus be used to evaluate not just bonds but any instrument that can be reduced to a deterministic set of cashflows.

ytm uses Newton's method to compute the yield-to-maturity of a bond (a.k.a. internal interest rate). When used with a bond, the initial outlay (i.e. the bonds dirty price) needs be included in the vector of cashflows. For a coupon bond, a good starting value y0 is the coupon divided by the dirty price of the bond.

An offset can be specified either as a single number or as a vector of zero rates. See Examples.

Value

Numeric.

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

See Also

NS, NSS

Examples

```
## ytm
cf <- c(5, 5, 5, 5, 5, 105)   ## cashflows
times <- 1:6                  ## maturities
y <- 0.0127                   ## the "true" yield
b0 <- vanillaBond(cf, times, yields = y)
cf <- c(-b0, cf); times <- c(0, times)
ytm(cf, times)

## ... with offset
cf <- c(5, 5, 5, 5, 5, 105)   ## cashflows
times <- 1:6                  ## maturities
y <- 0.02 + 0.01              ## risk-free 2% + risk-premium 1%
b0 <- vanillaBond(cf, times, yields = y)
cf <- c(-b0, cf); times <- c(0, times)
ytm(cf, times, offset = 0.02) ## ... only the risk-premium

cf <- c(5, 5, 5, 5, 5, 105)   ## cashflows
times <- 1:6                  ## maturities
y <- NS(c(6,9,10,5)/100, times) ## risk-premium 1%
b0 <- vanillaBond(cf, times, yields = y + 0.01)
cf <- c(-b0, cf); times <- c(0, times)
ytm(cf, times, offset = c(0,y)) ## ... only the risk-premium

## bonds
cf <- c(5, 5, 5, 5, 5, 105)   ## cashflows
times <- 1:6                  ## maturities
df <- 1/(1+y)^times           ## discount factors
all.equal(vanillaBond(cf, times, df),
          vanillaBond(cf, times, yields = y))

## ... using Nelson--Siegel
vanillaBond(cf, times, yields = NS(c(0.03,0,0,1), times))

## several bonds
##   cashflows are numeric vectors in a list 'cf',
##   times-to-payment are are numeric vectors in a
##   list 'times'

times <- list(1:3,
              1:4,
              0.5 + 0:5)
cf <- list(c(6, 6,          106),
```

```
           c(4, 4, 4,       104),
           c(2, 2, 2, 2, 2, 102))

alltimes <- sort(unique(unlist(times)))
M <- array(0, dim = c(length(cf), length(alltimes)))
for (i in seq_along(times))
    M[i, match(times[[i]], alltimes)] <- cf[[i]]
rownames(M) <- paste("bond.", 1:3, sep = "")
colnames(M) <- format(alltimes, nsmall = 1)

vanillaBond(cf = M, times = alltimes, yields = 0.02)

## duration/convexity
cf <- c(5, 5, 5, 5, 5, 105)   ## cashflows
times <- 1:6                  ## maturities
y <- 0.0527                   ## yield to maturity

d <- 0.001                   ## change in yield (+10 bp)
vanillaBond(cf, times, yields = y + d) - vanillaBond(cf, times, yields = y)

duration(cf, times, yield = y, raw = TRUE) * d

duration(cf, times, yield = y, raw = TRUE) * d +
    convexity(cf, times, yield = y, raw = TRUE)/2 * d^2
```

---

vanillaOptionEuropean *Pricing Plain-Vanilla Options (European and American)*

Description

Functions to calculate the theoretical prices and (some) Greeks for plain vanilla options.

Usage

```
vanillaOptionEuropean(S, X, tau, r, q, v, tauD = 0, D = 0,
                      type = "call", greeks = TRUE,
                      model = NULL, ...)
vanillaOptionAmerican(S, X, tau, r, q, v, tauD = 0, D = 0,
                      type = "call", greeks = TRUE, M = 101)

vanillaOptionImpliedVol(exercise = "european", price, S, X, tau, r,
                        q = 0, tauD = 0, D = 0, type = "call",
                        M = 101,
                        uniroot.control = list(), uniroot.info = FALSE)
```

Arguments

S	spot
X	strike
tau	time-to-maturity in years
r	risk-free rate
q	continuous dividend yield, see Details.
v	variance (volatility squared)
tauD	vector of times-to-dividends in years. Only dividends with tauD greater than zero and not greater than tau are kept.
D	vector of dividends (in currency units); default is no dividends.
type	call or put; default is call.
greeks	compute Greeks? Defaults to TRUE. But see Details for American options.
model	what model to use to value the option. Default is NULL which is equivalent to bsm.
...	parameters passed to pricing model
M	number of time steps in the tree
exercise	european (default) or american
price	numeric; the observed price to be recovered through choice of volatility.
uniroot.control	A list. If there are elements named interval, tol or maxiter, these are passed to uniroot. Any other elements of the list are ignored.
uniroot.info	logical; default is FALSE. If TRUE, the function will return the information returned by uniroot. See paragraph Value below.

Details

For European options the formula of Messrs Black, Scholes and Merton is used. It can be used for equities (set q equal to the dividend yield), futures (Black, 1976; set q equal to r), currencies (Garman and Kohlhagen, 1983; set q equal to the foreign risk-free rate). For future-style options (e.g. options on the German Bund future), set q and r equal to zero.

The Greeks are provided in their raw ('textbook') form with only one exception: Theta is made negative. For practical use, the other Greeks are also typically adjusted: Theta is often divided by 365 (or some other yearly day count); Vega and Rho are divided by 100 to give the sensitivity for one percentage-point move in volatility/the interest rate. Raw Gamma is not much use if not adjusted for the actual move in the underlier.

For European options the Greeks are computed through the respective analytic expressions. For American options only Delta, Gamma and Theta are computed because they can be directly obtained from the binomial tree; other Greeks need to be computed through a finite difference (see Examples).

For the European-type options, the function understands vectors of inputs, except for dividends. American options are priced via a Cox-Ross-Rubinstein tree; no vectorisation is implemented here.

The implied volatility is computed with uniroot from the **stats** package (the default search interval is c(0.00001,2); it can be changed through uniroot.control).

Dividends (D) are modelled via the escrowed-dividend model.

Value

Returns the price (a numeric vector of length one) if `greeks` is `FALSE`, else returns a list.

Note

If `greeks` is `TRUE`, the function will return a list with named elements (`value`, `delta` and so on). Prior to version 0.26-3, the first element of this list was named `price`.

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

See Also

`EuropeanCall`, `callCF`

Examples

```
S <- 100; X <- 100; tau <- 1; r <- 0.02; q <- 0.06; vol <- 0.3
unlist(vanillaOptionEuropean(S, X, tau, r, q, vol^2, type = "put"))

S <- 100; X <- 110; tau <- 1; r <- 0.1; q <- 0.06; vol <- 0.3; type <- "put"
unlist(vanillaOptionAmerican(S, X, tau, r, q, vol^2, type = type,
                             greeks = TRUE))

## compute rho for 1% move
h <- 0.01
(vanillaOptionAmerican(S, X, tau, r + h, q, vol^2,
    type = type, greeks = FALSE) -
 vanillaOptionAmerican(S, X, tau, r, q, vol^2,
    type = type, greeks = FALSE)) / (h*100)

## compute vega for 1% move
h <- 0.01
(vanillaOptionAmerican(S, X, tau, r, q,(vol + h)^2,
    type = type, greeks = FALSE) -
 vanillaOptionAmerican(S, X, tau, r, q, vol^2,
    type = type, greeks = FALSE)) / (h*100)



S <- 100; X <- 100
tau <- 1; r <- 0.05; q <- 0.00
D <- c(1,2); tauD <- c(0.3,.6)
```

```
type <- "put"
v <- 0.245^2  ## variance, not volatility

p <- vanillaOptionEuropean(S = S, X = X, tau, r, q, v = v,
                           tauD = tauD, D = D, type = type, greeks = FALSE)
vanillaOptionImpliedVol(exercise = "european", price = p,
     S = S, X = X, tau = tau, r = r, q = q, tauD = tauD, D = D, type = type)

p <- vanillaOptionAmerican(S = S, X = X, tau, r, q, v = v,
     tauD = tauD, D = D, type = type, greeks = FALSE)
vanillaOptionImpliedVol(exercise = "american", price = p,
     S = S, X = X, tau = tau, r = r, q = q, tauD = tauD, D = D, type =
     type, uniroot.control = list(interval = c(0.01, 0.5)))


## compute implied q
S <- 100; X <- 100
tau <- 1; r <- 0.05; q <- 0.072
v <- 0.22^2  ## variance, not volatility

call <- vanillaOptionEuropean(S=S, X = X, tau=tau, r=r, q=q, v=v,
         type = "call", greeks = FALSE)
put  <- vanillaOptionEuropean(S=S, X = X, tau=tau, r=r, q=q, v=v,
         type = "put", greeks = FALSE)

# ... the simple way
-(log(call + X * exp(-tau*r) - put) - log(S)) / tau

# ... the complicated way :-)
volDiffCreate <- function(exercise, call, put, S, X, tau, r) {
    f <- function(q) {
        cc <- vanillaOptionImpliedVol(exercise = exercise, price = call,
            S = S, X = X, tau = tau, r = r, q = q, type = "call")
        pp <- vanillaOptionImpliedVol(exercise = exercise, price = put,
            S = S, X = X, tau = tau, r = r, q = q, type = "put")
        abs(cc - pp)
    }
    f
}
f <- volDiffCreate(exercise = "european",
                   call = call, put = put, S = S, X = X, tau = tau, r)
optimise(f,interval = c(0, 0.2))$minimum

##
S <- 100; X <- 100
tau <- 1; r <- 0.05; q <- 0.072
v <- 0.22^2  ## variance, not volatility
vol <- 0.22

vanillaOptionEuropean(S=S, X = X, tau=tau, r=r, q=q, v=v,     ## with variance
                      type = "call", greeks = FALSE)
vanillaOptionEuropean(S=S, X = X, tau=tau, r=r, q=q, vol=vol, ## with vol
                      type = "call", greeks = FALSE)
```

```
vanillaOptionEuropean(S=S, X = X, tau=tau, r=r, q=q, vol=vol, ## with vol
                      type = "call", greeks = FALSE, v = 0.2^2)
```

`xtContractValue` *Contract Value of Australian Government Bond Future*

Description

Compute the contract value of an Australian government-bond future from its quoted price.

Usage

```
xtContractValue(quoted.price, coupon, do.round = TRUE)
xtTickValue(quoted.price, coupon, do.round = TRUE)
```

Arguments

`quoted.price`	The price, as in `99.02`.
`coupon`	numeric; should be 6, not 0.06
`do.round`	If `TRUE`, round as done by ASX clearing house.

Details

Australian government-bond futures, traded at the Australian Securities Exchange (ASX), are quoted as `100 - yield`. The function computes the actual contract value from the quoted price.

`xtTickValue` computes the tick value via a central difference.

Value

A numeric vector.

Author(s)

Enrico Schumann

References

http://www.rba.gov.au/mkt-operations/resources/tech-notes/pricing-formulae.html

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

Examples

```
quoted.price <- 99
coupon <- 6
xtContractValue(quoted.price, coupon)
xtTickValue(quoted.price, coupon)
## convexity
quoted.price <- seq(90, 100, by = 0.1)
plot(100 - quoted.price,
     xtContractValue(quoted.price, coupon),
     xlab = "Yield", ylab = "Contract value")
```

---

xwGauss *Integration of Gauss-type*

Description

Compute nodes and weights for Gauss integration.

Usage

```
xwGauss(n, method = "legendre")
changeInterval(nodes, weights, oldmin, oldmax, newmin, newmax)
```

Arguments

n	number of nodes
method	character. default is "legendre"; also possible are "laguerre" and "hermite"
nodes	the nodes (a numeric vector)
weights	the weights (a numeric vector)
oldmin	the minimum of the interval (typically as tabulated)
oldmax	the maximum of the interval (typically as tabulated)
newmin	the desired minimum of the interval
newmax	the desired maximum of the interval

Details

xwGauss computes nodes and weights for integration for the interval -1 to 1. It uses the method of Golub and Welsch (1969).

changeInterval is a utility that transforms nodes and weights to an arbitrary interval.

Value

a list with two elements

weights	a numeric vector
nodes	a numeric vector

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2011) *Numerical Methods and Optimization in Finance*. Elsevier. http://www.elsevierdirect.com/product.jsp?isbn=9780123756626

Golub, G.H. and Welsch, J.H. (1969). Calculation of Gauss Quadrature Rules. *Mathematics of Computation*, **23**(106), pp. 221–230+s1–s10.

Schumann, E. (2016) Financial Optimisation with R (**NMOF** Manual). http://enricoschumann.net/NMOF.htm#NMOFmanual

See Also

callHestoncf

Examples

```
## examples from Gilli/Maringer/Schumann (2011), ch. 15

## a test function
f1 <- function(x) exp(-x)
m <- 5; a <- 0; b <- 5
h <- (b - a)/m

## rectangular rule -- left
w <- h; k <- 0:(m-1); x <- a + k * h
sum(w * f1(x))

## rectangular rule -- right
w <- h; k <- 1:m ; x <- a + k * h
sum(w * f1(x))

## midpoint rule
w <- h; k <- 0:(m-1); x <- a + (k + 0.5)*h
sum(w * f1(x))

## trapezoidal rule
w <- h
k <- 1:(m-1)
x <- c(a, a + k*h, b)
aux <- w * f1(x)
sum(aux) - (aux[1] + aux[length(aux)])/2

## R's integrate (from package stats)
integrate(f1, lower = a,upper = b)

## Gauss--Legendre
temp <- xwGauss(m)
temp <- changeInterval(temp$nodes, temp$weights,
                       oldmin = -1, oldmax = 1, newmin =  a, newmax = b)
```

```
x <- temp$nodes; w <- temp$weights
sum(w * f1(x))
```

Index

∗Topic **datagen**
resampleC, 57
∗Topic **datasets**
bundData, 6
fundData, 25
optionData, 45
∗Topic **distribution**
resampleC, 57
∗Topic **optimize**
bracketing, 5
DEopt, 18
GAopt, 26
gridSearch, 29
LSopt, 33
PSopt, 48
SAopt, 62
TAopt, 69
testFunctions, 74
∗Topic **package**
NMOF-package, 3
.Random.seed, *19*, *28*, *34*, *50*, *64*, *71*

bracketing, *3*, 5
bundData, *3*, 6
bundFuture, *3*, 7
bundFutureImpliedRate *(*bundFuture*)*, 7

callCF, *3*, 9, *13, 14*, *81*
callHestoncf, *10*, 11, *25*, *85*
callMerton, 13
cat, *54*
cfBates *(*callCF*)*, 9
cfBSM *(*callCF*)*, 9
cfHeston *(*callCF*)*, 9
cfMerton *(*callCF*)*, 9
cfVG *(*callCF*)*, 9
changeInterval *(*xwGauss*)*, 84
colSubset, 15
convexity *(*vanillaBond*)*, 76
CPPI, 16

data.frame, *67*
Date, *8*
DEopt, *3*, 18, *27, 28*, *50*, *59*, *76*
divRatio, 21
drawdown, 22
duration *(*vanillaBond*)*, 76

EuropeanCall, *13, 14*, 24, *81*
EuropeanCallBE *(*EuropeanCall*)*, 24
expand.grid, *30*

file.show, *66*
fundData, *3*, 25

GAopt, *3*, *20*, 26, *59*
gbb *(*mc*)*, 38
gbm *(*mc*)*, 38
grepl, *66*
gridSearch, *3*, 29

lapply, *30*
list, *54*
LS.info, 31
LSopt, *3*, *31, 32*, 33, *59*, *64*, *72*

MA, 37
mc, *3*, 38
minvar, 40

NMOF *(*NMOF-package*)*, 3
NMOF-package, 3
NS, 42, *45*, *78*
NSf, *43*, 44
NSS, *45*, *78*
NSS *(*NS*)*, 42
NSSf, *43*
NSSf *(*NSf*)*, 44

optionData, *3*, 45

parent.frame, *32*, *61*, *68*

pm, 46
pretty, *54*
PSopt, *3*, *20*, *27, 28*, 48, *59*, *76*
putCallParity, *3*, *10*, *14*, 51

qr, *15*
qTable, 53

repairMatrix, *16*, 55, *57*
resampleC, *3*, 57
restartOpt, *3*, *34*, 58, *64*, *72*

SA.info, 60
SAopt, *61*, 62
showChapterNames *(*showExample*)*, 66
showExample, 66
solve.QP, *41*

TA.info, *32*, *61*, 67
TAopt, *3*, *34*, *41*, *59*, *64*, *67, 68*, 69
testFunctions, 74
tfAckley *(*testFunctions*)*, 74
tfEggholder *(*testFunctions*)*, 74
tfGriewank *(*testFunctions*)*, 74
tfRastrigin *(*testFunctions*)*, 74
tfRosenbrock *(*testFunctions*)*, 74
tfSchwefel *(*testFunctions*)*, 74
tfTrefethen *(*testFunctions*)*, 74
txtProgressBar, *19*, *63*, *71*

uniroot, *6*, *10*, *12*, *14*, *80*

vanillaBond, *3*, 76
vanillaOptionAmerican, *3*
vanillaOptionAmerican
        *(*vanillaOptionEuropean*)*, 79
vanillaOptionEuropean, *3*, *39*, *52*, 79
vanillaOptionImpliedVol
        *(*vanillaOptionEuropean*)*, 79

xtContractValue, *3*, 83
xtTickValue *(*xtContractValue*)*, 83
xwGauss, 84

ytm *(*vanillaBond*)*, 76