Package: adagio (via r-universe)

August 21, 2024

Type Package

Title Discrete and Global Optimization Routines

Version 0.9.2

Date 2023-10-26

Maintainer Hans W. Borchers <hwborchers@googlemail.com>

Depends R (>= 3.1.0)

Imports graphics, stats, lpSolve (>= 5.6.15)

Description The R package 'adagio' will provide methods and algorithms for (discrete) optimization, e.g. knapsack and subset sum procedures, derivative-free Nelder-Mead and Hooke-Jeeves minimization, and some (evolutionary) global optimization functions.

License GPL (>= 3)

NeedsCompilation no

Author Hans W. Borchers [aut, cre]

Date/Publication 2023-10-26 13:30:03 UTC

Repository https://hwborchers.r-universe.dev

RemoteUrl https://github.com/cran/adagio

RemoteRef HEAD

RemoteSha ff8105f5a43467efb76d72f264cecc844037f70e

Contents

`assignment` *Linear Sum Assignment Problem*

Description

Linear (sum) assignment problem, or LSAP.

Usage

```
assignment(cmat, dir = "min")
```

Arguments

`cmat`	quadratic (numeric) matrix, the cost matrix.
`dir`	direction, can be "min" or "max".

Details

Solves the linear (sum) assignment problem for quadratic matrices. Uses the `lp.assign` function from the `lpSolve` package, that is it solves LSAP as a mixed integer linear programming problem.

Value

List with components `perm`, the permutation that defines the minimum solution, `min`, the minimum value, and `err` is always `0`, i.e. not used at the moment.

Note

Slower than the Hungarian algorithm in package `clue`.

References

Burkard, R., M. Dell'Amico, and S. Martello (2009). Assignment Problems. Society for Industrial and Applied Mathematics (SIAM).

Martello, S., and P. Toth (1990). Knapsack Problems: Algorithms and Computer Implementations. John Wiley & Sons, Ltd.

See Also

`clue::solve_LSAP`

Examples

```
##  Example similar to clue::solve_LSAP
set.seed(8237)
x <- matrix(sample(1:100), nrow = 10)
y <- assignment(x)
# show permutation and check minimum sum
y$perm                        #   7  6 10  5  8  2  1  4  9  3
y$min                         # 173
z <- cbind(1:10, y$perm)
x[z]                          #  16  9 49  6 17 14  1 44 10  7
y$min == sum(x[z])            # TRUE

## Not run:
##  Example: minimize sum of distances of complex points
n <- 100
x <- rt(n, df=3) + 1i * rt(n, df=3)
y <- runif(n) + 1i * runif(n)
cmat <- round(outer(x, y, FUN = function(x,y) Mod(x - y)), 2)
system.time(T1 <- assignment(cmat))     # elapsed: 0.003
T1$min / 100                            # 145.75

## Hungarian algorithm in package 'clue'
library("clue")
system.time(T2 <- solve_LSAP(cmat))     # elapsed: 0.014
sum(cmat[cbind(1:n, T2)])               # 145.75

## End(Not run)
```

bpp_approx *Approximate Bin Packing*

Description

Solves the Bin Packing problem approximately.

Usage

```
bpp_approx(S, cap, method = c("firstfit", "bestfit", "worstfit"))
```

Arguments

`S`	vector of weights (or sizes) of items.
`cap`	same capacity for all the bins.
`method`	which approximate method to use.

Details

Solves approximately the Bin Packing problem for numeric weights and bins, all having the same volume.

Possible methods are "firstfit", "bestfit", and "worstfit". "firstfit" tries to place each item as early as possible, "bestfit" such that the remaining space in the bin is as small as possible, and "worstfit" such that the remaining space is as big as possible.

Best results are achieved with the "bestfit" method. "firstfit" may be a reasonable alternative. For smaller and medium-sized data the approximate results will come quite close to the exact solution, see the examples.

In general, the results are much better if the items in `S` are sorted decreasingly. If they are not, an immediate warning is issued.

Value

A list of the following components:

`nbins`	minimum number of bins.
`xbins`	index of the bin each item is assigned to.
`sbins`	sum of item sizes in each bin.
`filled`	total volume filled in the bins (as percentage).

Note

The Bin Packing problem can be solved as a Linear Program. The formulation is a bit tricky, and it turned out 'lpSolve' does not solve medium-sized problems in acceptable time. (Tests with 'Rglpk' will follow.)

Author(s)

Hans W. Borchers

References

Silvano Martello. "Bin packing problems". In: 23rd Belgian Mathematical Optimization Workshop, La-Roche-en-Ardennes 2019.

See Also

Function `binpacking` in package 'knapsack' (on R-Forge).

Examples

```
## (1)
S <- c(50, 3, 48, 53, 53, 4, 3, 41, 23, 20, 52, 49)
cap <- 100
bpp_approx(S, cap, method = "bestfit")
## exact    -- $nbins 4, filled 99.75 %
## firstfit -- $nbins 6, filled 66.5  %
## bestfit  -- $nbins 5, filled 79.8  %
## ! when decreasingly sorted, 'bestfit' with nbins = 4

## (2)
S <- c(100,99,89,88,87,75,67,65,65,57,57,49,47,31,27,18,13,9,8,1)
cap <- 100
bpp_approx(S, cap, method = "firstfit")
# firstfit: 12 bins; exact: 12 bins

## Not run:
## (3)
S <-  c(99,99,96,96,92,92,91,88,87,86,
        85,76,74,72,69,67,67,62,61,56,
        52,51,49,46,44,42,40,40,33,33,
        30,30,29,28,28,27,25,24,23,22,
        21,20,17,14,13,11,10, 7, 7, 3)
cap <- 100
bpp_approx(S, cap)
# exact: 25; firstfit: 25; bestfit: 25 nbins

## (4)
# 20 no.s in 1..100, capacity 100
set.seed(7013)
S <- sample(1:100, 20, replace = TRUE)
cap <- 100
bpp_approx(sort(S, decreasing = TRUE), cap, method = "bestfit")
# exact: 12 bins; firstfit and bestfit: 13; worstfit: 14 bins

## End(Not run)
```

`change_making` *Change Making Problem*

Description

Solves the Change Making problem as an integer linear program.

Usage

```
change_making(items, value)
```

Arguments

`items`	vector of integer numbers greater than zero.
`value`	integer number

Details

The Change Making problem attempts to find a minimal combination of items that sum up to a given value. If the items are distinct positive integers, the solution is unique.

If the problem is infeasible, i.e. there is no such combination, the returned `count` is 0.

The problem is treated as an Integer Linear Program (ILP) and solved with the `lp` solver in `lpSolve`.

Value

Returns a list with components `count`, the number of items used to sum up to the value, and `solution`, the number of items used per item.

References

See the Wikipedia article on the "change making problem".

See Also

`setcover`

Examples

```
items = c(2, 5, 10, 50, 100)
value = 999
change_making(items, value)

## Not run:
solutions <- numeric(20)
for (m in 1:20) {
    sol <- change_making(items, m)
    solutions[m] <- sol$count
}
solutions
#>  [1] 0 1 0 2 1 3 2 4 3 1 4 2 5 3 2 4 3 5 4 2

## End(Not run)
```

`CMAES` *Covariance Matrix Adaptation Evolution Strategy*

Description

The CMA-ES (Covariance Matrix Adaptation Evolution Strategy) is an evolutionary algorithm for difficult non-linear non-convex optimization problems in continuous domain. The CMA-ES is typically applied to unconstrained or bounded constraint optimization problems, and search space dimensions between three and fifty.

Usage

```
pureCMAES(par, fun, lower = NULL, upper = NULL, sigma = 0.5,
                    stopfitness = -Inf, stopeval = 1000*length(par)^2, ...)
```

Arguments

`par`	objective variables initial point.
`fun`	objective/target/fitness function.
`lower`, `upper`	lower and upper bounds for the parameters.
`sigma`	coordinate wise standard deviation (step size).
`stopfitness`	stop if fitness < stopfitness (minimization).
`stopeval`	stop after stopeval number of function evaluations
`...`	additional parameters to be passed to the function.

Details

The CMA-ES implements a stochastic variable-metric method. In the very particular case of a convex-quadratic objective function the covariance matrix adapts to the inverse of the Hessian matrix, up to a scalar factor and small random fluctuations. The update equations for mean and covariance matrix maximize a likelihood while resembling an expectation-maximization algorithm.

Value

Returns a list with components `xmin` and `fmin`.

Be patient; for difficult problems or high dimensions the function may run for several minutes; avoid problem dimensions of 30 and more!

Note

There are other implementations of Hansen's CMAES in package 'cmaes' (simplified form) and in package 'parma' as cmaes() (extended form).

Author(s)

Copyright (c) 2003-2010 Nikolas Hansen for Matlab code PURECMAES; converted to R by Hans W Borchers. (Hansen's homepage: www.cmap.polytechnique.fr/~nikolaus.hansen/)

References

Hansen, N. (2011). The CMA Evolution Strategy: A Tutorial.
https://arxiv.org/abs/1604.00772

Hansen, N., D.V. Arnold, and A. Auger (2013). Evolution Strategies. J. Kacprzyk and W. Pedrycz (Eds.). Handbook of Computational Intelligence, Springer-Verlag, 2015.

See Also

`cmaes::cmaes`, `parma::cmaes`

Examples

```
## Not run:
##  Polynomial minimax approximation of data points
##  (see the Remez algorithm)
n <- 10; m <- 101           # polynomial of degree 10; no. of data points
xi <- seq(-1, 1, length = m)
yi <- 1 / (1 + (5*xi)^2)    # Runge's function

pval <- function(p, x)      # Horner scheme
    outer(x, (length(p) - 1):0, "^") %*% p

pfit <- function(x, y, n)   # polynomial fitting of degree n
    qr.solve(outer(x, seq(n, 0), "^"), y)

fn1 <- function(p)           # objective function
    max(abs(pval(p, xi) - yi))

pf <- pfit(xi, yi, 10)      # start with a least-squares fitting
sol1 <- pureCMAES(pf, fn1, rep(-200, 11), rep(200, 11))
zapsmall(sol1$xmin)
# [1]  -50.24826    0.00000  135.85352    0.00000 -134.20107    0.00000
# [7]   59.19315    0.00000  -11.55888    0.00000    0.93453

print(sol1$fmin, digits = 10)
# [1] 0.06546780411

##  Polynomial fitting in the L1 norm
##  (or use LP or IRLS approaches)
fn2 <- function(p)
    sum(abs(pval(p, xi) - yi))

sol2 <- pureCMAES(pf, fn2, rep(-100, 11), rep(100, 11))
zapsmall(sol2$xmin)
# [1] -21.93238   0.00000  62.91083   0.00000 -67.84847   0.00000
# [7]  34.14398   0.00000  -8.11899   0.00000   0.84533

print(sol2$fmin, digits = 10)
# [1] 3.061810639

## End(Not run)
```

`fminviz,flineviz` *Visualize Function Minimum*

Description

Visualizes multivariate functions around a point or along a line between two points in `R^n`.

Usage

```
fminviz(fn, x0, nlines = 2*length(x0),
        npoints = 51, scaled = 1.0)

flineviz(fn, x1, x2, npoints = 51, scaled = 0.1)
```

Arguments

`fn`	multivariate function to be visualized.
`x0, x1, x2`	points in n-dimensional space.
`nlines`	number of lines to plot.
`npoints`	number of points used to plot a line.
`scaled`	scale factor to extend the line(s).

Details

`fminviz` vizualizes the behavior of a multivariate function `fn` around a point `x0`. It randomly selects `nlines` lines through `x0` in `R^n` and draws the curves of the function along these lines in one graph.

Curves that have at least one point below `fn(x0)` are drawn in red, all others in blue. The scale on the x-axis is the Euclidean distance in `R^n`. The `scale` factor can change it.

`flineviz` vizualizes the behavior of a multivariate function `fn` along the straight line between the points `x1` and `x2`. Points `x1` and `x2` are also plotted.

Value

Plots a line graph and returns `NULL` (invisibly).

Examples

```
## Not run:
  f1 <- function(x) x[1]^2 - x[2]^2
  fminviz(f1, c(0, 0), nlines = 10)

  f2 <- function(x) (1 - x[1])^2 + 100*(x[2] - x[1]^2)^2
  flineviz(f2, c(0, 0), c(1, 1))

## End(Not run)
```

`hamiltonian` *Finds a Hamiltonian path or cycle*

Description

A Hamiltionian path or cycle (a.k.a. Hamiltonian circuit) is a path through a graph that visits each vertex exactly once, resp. a closed path through the graph.

Usage

```
hamiltonian(edges, start = 1, cycle = TRUE)
```

Arguments

`edges`	an edge list describing an undirected graph.
`start`	vertex number to start the path or cycle.
`cycle`	Boolean, should a path or a full cycle be found.

Details

`hamiltonian()` applies a backtracking algorithm that is relatively efficient for graphs of up to 30–40 vertices. The edge list is first transformed to a list where the `i`-th component contains the list of all vertices connected to vertex `i`.

The edge list must be of the form `c(v1, v2, v3, v2, ...)` meaning that there are edges `v1 --> v2`, `v3 --> v4`, etc., connecting these vertices. Therefore, an edge list has an even number of entries.

If the function returns `NULL`, there is no Hamiltonian path or cycle. The function does not check if the graph is connected or not. And if `cycle = TRUE` is used, then there also exists an edge from the last to the first entry in the resulting path.

Ifa Hamiltonian cycle exists in the graph it will be found whatever the starting vertex was. For a Hamiltonian path this is different and a successful search may very well depend on the start.

Value

Returns a vector containing vertex number of a valid path or cycle, or `NULL` if no path or cycle has been found (i.e., does not exist); If a cycle was requested, there exists an edge from the last to the first vertex in this list of edges.

Note

See the `igraph` package for more information about handling graphs and defining them through edge lists or other constructs.

Author(s)

Hans W. Borchers

References

Papadimitriou, Ch. H., and K. Steiglitz (1998). Optimization Problems: Algorithms and Complexity. Prentice-Hall/Dover Publications.

See Also

Package `igraph`

Examples

```
## Dodekaeder graph
D20_edges <- c(
     1,  2,  1,  5,  1,  6,  2,  3,  2,  8,  3,  4,  3, 10,  4,  5,  4, 12,
     5, 14,  6,  7,  6, 15,  7,  8,  7, 16,  8,  9,  9, 10,  9, 17, 10, 11,
    11, 12, 11, 18, 12, 13, 13, 14, 13, 19, 14, 15, 15, 20, 16, 17, 16, 20,
    17, 18, 18, 19, 19, 20)
hamiltonian(D20_edges, cycle = TRUE)
# [1]  1  2  3  4  5 14 13 12 11 10  9  8  7 16 17 18 19 20 15  6
hamiltonian(D20_edges, cycle = FALSE)
# [1]  1  2  3  4  5 14 13 12 11 10  9  8  7  6 15 20 16 17 18 19

## Herschel graph
# The Herschel graph the smallest non-Hamiltonian polyhedral graph.
H11_edges <- c(
     1,  2,  1,  8,  1,  9,  1, 10,  2,  3,  2, 11,  3,  4,  3,  9,  4,  5,
     4, 11,  5,  6,  5,  9,  5, 10,  6,  7,  6, 11,  7,  8,  7, 10,  8, 11)
hamiltonian(H11_edges, cycle = FALSE)
# NULL

## Not run:
## Example: Graph constructed from squares
N <- 45  # 23, 32, 45
Q <- (2:trunc(sqrt(2*N-1)))^2
sq_edges <- c()
for (i in 1:(N-1)) {
    for (j in (i+1):N) {
        if ((i+j)
            sq_edges <- c(sq_edges, i, j)
    }
}

require(igraph)
sq_graph <- make_graph(sq_edges, directed=FALSE)
plot(sq_graph)

if (N == 23) {
    # does not find a path with start=1 ...
    hamiltonian(sq_edges, start=18, cycle=FALSE)
    # hamiltonian(sq_edges)                     # NULL
} else if (N == 32) {
    # the first of these graphs that is Hamiltonian ...
    # hamiltonian(sq_edges, cycle=FALSE)
    hamiltonian(sq_edges)
} else if (N == 45) {
    # takes much too long ...
    # hamiltonian(sq_edges, cycle=FALSE)
    hamiltonian(sq_edges)
}
## End(Not run)
```

Historize *Historize function*

Description

Provides storage for function calls.

Usage

```
Historize(fun, len = 0, ...)
```

Arguments

`fun`	Function of one or several variables.
`len`	If > 0, input values will be stored, too.
`...`	Additional parameters to be handed to the function.

Details

Historize() adds storage to the function. If `len=0` then only function values will be stored in a vector, if `len>0` then variables will be stored in a vector, and function values will be stored in a matrix, one line for every function call.

If `Fn` = `Historize(...)` is the 'historized' function, then the storage can be retrieved with `Fn()`, and can be cleared with `Fn(NULL)`.

Filling the storage will take extra time and can slow down the function calls. Especially also storing the variables used in the call (with `len>0`) will make it considerably slower.

Functions can have multivariate output; the user is asked to take care of handling the output vector or matrix correctly. The function may even require additional parameters.

Value

Returns a list, $input the input variables as matrix, $H2 the function values as vector, $nvars the number of input variables of the function, and $ncalls the number or recorded function calls.

Note

Can also be applied to functions that output a vector (same length for every call).

Author(s)

Hans W. Borchers

See Also

`trace`

Examples

```
f <- function(x) sum(x^2)
F <- Historize(f, len = 1)
c( F(c(1,1)), F(c(1,2)), F(c(2,1)), F(c(2,2)) )
#> [1] 2 5 5 8
F()
#> $input
#>      [,1] [,2]
#> [1,]    1    1
#> [2,]    1    2
#> [3,]    2    1
#> [4,]    2    2
#>
#> $values
#> [1] 2 5 5 8
#>
#> $nvars
#> [1] 2
#>
#> $ncalls
#> [1] 4
F(NULL)     # reset memory


## Rastrigin under Differential Evolution

Fn <- Historize(fnRastrigin)
dm <- 10
lb <- rep(-5.2, dm); ub <- -lb

sol <- simpleDE(Fn, lower = lb, upper = ub)
fvalues <- Fn()$values
fvals <- cummin(fvalues)

## Not run:
plot(fvalues, type = 'p', col = 7, pch = '.', cex = 2,
     main = "Simple DE optimization", xlab = '', ylab = '')
lines(fvals, col = 2, lwd = 2)
legend(length(fvalues), max(fvalues),
       c("Intermediate values", "cummulated min"),
       xjust = 1, col = c(7, 2), lwd = 2)
grid()
## End(Not run)
```

hookejeeves *Hooke-Jeeves Minimization Method*

Description

An implementation of the Hooke-Jeeves algorithm for derivative-free optimization.

Usage

```
hookejeeves(x0, f, lb = NULL, ub = NULL,
            tol = 1e-08,
            target = Inf, maxfeval = Inf, info = FALSE, ...)
```

Arguments

`x0`	starting vector.
`f`	nonlinear function to be minimized.
`lb`, `ub`	lower and upper bounds.
`tol`	relative tolerance, to be used as stopping rule.
`target`	iteration stops when this value is reached.
`maxfeval`	maximum number of allowed function evaluations.
`info`	logical, whether to print information during the main loop.
`...`	additional arguments to be passed to the function.

Details

This method computes a new point using the values of `f` at suitable points along the orthogonal coordinate directions around the last point.

Value

List with following components:

`xmin`	minimum solution found so far.
`fmin`	value of `f` at minimum.
`fcalls`	number of function evaluations.
`niter`	number of iterations performed.

Note

Hooke-Jeeves is notorious for its number of function calls. Memoization is often suggested as a remedy.

For a similar implementation of Hooke-Jeeves see the 'dfoptim' package.

References

C.T. Kelley (1999), Iterative Methods for Optimization, SIAM.

Quarteroni, Sacco, and Saleri (2007), Numerical Mathematics, Springer-Verlag.

See Also

`neldermead`

Examples

```
##  Rosenbrock function
rosenbrock <- function(x) {
    n <- length(x)
    x1 <- x[2:n]
    x2 <- x[1:(n-1)]
    sum(100*(x1-x2^2)^2 + (1-x2)^2)
}

hookejeeves(c(0,0,0,0), rosenbrock)
# $xmin
# [1] 1.000000 1.000001 1.000002 1.000004
# $fmin
# [1] 4.774847e-12
# $fcalls
# [1] 2499
# $niter
#[1] 26

hookejeeves(rep(0,4), lb=rep(-1,4), ub=0.5, rosenbrock)
# $xmin
# [1] 0.50000000 0.26221320 0.07797602 0.00608027
# $fmin
# [1] 1.667875
# $fcalls
# [1] 571
# $niter
# [1] 26
```

`knapsack` *0-1 Knapsack Problem*

Description

Solves the 0-1 (binary) single knapsack problem.

Usage

```
knapsack(w, p, cap)
```

Arguments

`w`	integer vector of weights.
`p`	integer vector of profits.
`cap`	maximal capacity of the knapsack, integer too.

Details

`knapsack` solves the 0-1, or: binary, single knapsack problem by using the dynamic programming approach. The problem can be formulated as:

Maximize `sum(x*p)` such that `sum(x*w)` <= `cap`, where `x` is a vector with `x[i]` == `0` or `1`.

Knapsack procedures can even solve subset sum problems, see the examples 3 and 3' below.

Value

A list with components `capacity`, `profit`, and `indices`.

Author(s)

HwB email: <hwborchers@googlemail.com>

References

Papadimitriou, C. H., and K. Steiglitz (1998). Combinatorial Optimization: Algorithms and Complexity. Dover Publications 1982, 1998.

Horowitz, E., and S. Sahni (1978). Fundamentals of Computer Algorithms. Computer Science Press, Rockville, ML.

See Also

`knapsack::knapsack`

Examples

```
# Example 1
p <- c(15, 100, 90, 60, 40, 15, 10,  1)
w <- c( 2,  20, 20, 30, 40, 30, 60, 10)
cap <- 102
(is <- knapsack(w, p, cap))
# [1] 1 2 3 4 6 , capacity 102 and total profit 280

## Example 2
p <- c(70, 20, 39, 37, 7, 5, 10)
w <- c(31, 10, 20, 19, 4, 3,  6)
cap <- 50
(is <- knapsack(w, p, cap))
# [1] 1 4 , capacity 50 and total profit 107

## Not run:
##  Example 3: subset sum
```

```
p <- seq(2, 44, by = 2)^2
w <- p
is <- knapsack(w, p, 2012)
p[is$indices]  # 16  36  64 144 196 256 324 400 576

##  Example 3': maximize number of items
#   w <- seq(2, 44, by = 2)^2
#   p <- numeric(22) + 1
#  is <- knapsack(w, p, 2012)

## Example 4 from Rosetta Code:
w = c(  9, 13, 153,  50, 15, 68, 27, 39, 23, 52, 11,
       32, 24,  48,  73, 42, 43, 22,  7, 18,  4, 30)
p = c(150, 35, 200, 160, 60, 45, 60, 40, 30, 10, 70,
       30, 15,  10,  40, 70, 75, 80, 20, 12, 50, 10)
cap = 400
system.time(is <- knapsack(w, p, cap))  # 0.001 sec

## End(Not run)
```

maxempty — *Maximally Empty Rectangle Problem*

Description

Find the largest/maximal empty rectangle, i.e. with largest area, not containing given points.

Usage

```
maxempty(x, y, ax = c(0, 1), ay = c(0, 1))
```

Arguments

x, y	coordinates of points to be avoided.
ax, ay	left and right resp. lower and upper constraints.

Details

Find the largest or maximal empty two-dimensional rectangle in a rectangular area. The edges of this rectangle have to be parallel to the edges of the enclosing rectangle (and parallel to the coordinate axes). ‘Empty’ means that none of the points given are contained in the interior of the found rectangle.

Value

List with `area` and `rect` the rectangle as a vector usable for the `rect` graphics function.

Note

The algorithm has a run-time of `O(n^2)` while there are run-times of `O(n*log(n))` reported in the literature, utilizing a more complex data structure. I don't know of any comparable algorithms for the largest empty circle problem.

Author(s)

HwB email: <hwborchers@googlemail.com>

References

B. Chazelle, R. L. Drysdale, and D. T. Lee (1986). Computing the Largest Empty Rectangle. SIAM Journal of Computing, Vol. 15(1), pp. 300–315.

A. Naamad, D. T. Lee, and W.-L. Hsu (1984). On the Maximum Empty Rectangle Problem. Discrete Applied Mathematics, Vol. 8, pp. 267–277.

See Also

`Hmisc::largest.empty` with a Fortran implementation of this code.

Examples

```
N <- 100; set.seed(8237)
x <- runif(N); y <- runif(N)
R <- maxempty(x, y, c(0,1), c(0,1))
R
# $area
# [1] 0.08238793
# $rect
# [1] 0.7023670 0.1797339 0.8175771 0.8948442

## Not run:
plot(x, y, pch="+", xlim=c(0,1), ylim=c(0,1), col="darkgray",
      main = "Maximally empty rectangle")
rect(0, 0, 1, 1, border = "red", lwd = 1, lty = "dashed")
do.call(rect, as.list(R$rect))
grid()
## End(Not run)
```

---

maxquad *The MAXQUAD Test Function*

Description

Lemarechal's MAXQUAD optimization test function.

Usage

```
maxquad(n, m)
```

Arguments

`n`	number of variables of the generated test function.
`m`	number of functions to compete for the maximum.

Details

MAXQUAD actually is a family of minimax functions, parametrized by the number `n` of variables and the number `m` of functions whose maximum it is.

Value

Returns a list with components `fn` the generated test function of `n` variables, and `gr` the corresponding (analytical) gradient function.

References

Kuntsevich, A., and F. Kappel (1997). SolvOpt – The Solver for Local Nonlinear Optimization Problems. Manual Version 1.1, Institute of Mathematics, University of Graz.

Lemarechal, C., and R. Mifflin, Eds. (1978). Nonsmooth Optimization. Pergamon Press, Oxford.

Shor, N. Z. (1985). Minimization Methods for Non-differentiable Functions. Series in Computational Mathematics, Springer-Verlag, Berlin.

Examples

```
# Test function of 5 variables, defined as maximum of 5 smooth functions
maxq <- maxquad(5, 5)
fnMaxquad <- maxq$fn
grMaxquad <- maxq$gr
# shor
```

`maxsub` *Maximal Sum Subarray*

Description

Find a subarray with maximal positive sum.

Usage

```
maxsub(x, inds = TRUE)

maxsub2d(A)
```

Arguments

`x`	numeric vector.
`A`	numeric matrix
`inds`	logical; shall the indices be returned?

Details

`maxsub` finds a contiguous subarray whose sum is maximally positive. This is sometimes called Kadane's algorithm. `maxsub` will use a very fast version with a running time of `O(n)` where `n` is the length of the input vector `x`.

`maxsub2d` finds a (contiguous) submatrix whose sum of elements is maximally positive. The approach taken here is to apply the one-dimensional routine to summed arrays between all rows of `A`. This has a run-time of `O(n^3)`, though a run-time of `O(n^2 log n)` seems possible see the reference below. `maxsub2d` can solve a 100-by-100 matrix in a few seconds – but beware of bigger ones.

Value

Either just a maximal sum, or a list this sum as component `sum` plus the start and end indices as a vector `inds`.

Note

In special cases, the matrix `A` may be sparse or (as in the example section) only have one nonzero element in each row and column. Expectation is that there may exists a more efficient (say `O(n^2)`) algorithm in these special cases.

Author(s)

HwB <hwborchers@googlemail.com>

References

Bentley, Jon (1986). "Programming Pearls", Column 7. Addison-Wesley Publ. Co., Reading, MA.

T. Takaoka (2002). Efficient Algorithms for the Maximum Subarray Problem by Distance Matrix Multiplication. The Australasian Theory Symposion, CATS 2002.

Examples

```
##  Find a maximal sum subvector
set.seed(8237)
x <- rnorm(1e6)
system.time(res <- maxsub(x, inds = TRUE))
res

##  Standard example: Find a maximal sum submatrix
A <- matrix(c(0,-2,-7,0, 9,2,-6,2, -4,1,-4,1, -1,8,0,2),
            nrow = 4, ncol = 4, byrow =TRUE)
maxsub2d(A)
# $sum:  15
# $inds: 2 4 1 2 , i.e., rows = 2..4, columns = 1..2

## Not run:
##  Application to points in the unit square:
set.seed(723)
N <- 50; w <- rnorm(N)
x <- runif(N); y <- runif(N)
```

```
clr <- ifelse (w >= 0, "blue", "red")
plot(x, y, pch = 20, col = clr, xlim = c(0, 1), ylim = c(0, 1))

xs <- unique(sort(x)); ns <- length(xs)
X  <- c(0, ((xs[1:(ns-1)] + xs[2:ns])/2), 1)
ys <- unique(sort(y)); ms <- length(ys)
Y  <- c(0, ((ys[1:(ns-1)] + ys[2:ns])/2), 1)
abline(v = X, col = "gray")
abline(h = Y, col = "gray")

A <- matrix(0, N, N)
xi <- findInterval(x, X); yi <- findInterval(y, Y)
for (i in 1:N) A[yi[i], xi[i]] <- w[i]

msr <- maxsub2d(A)
rect(X[msr$inds[3]], Y[msr$inds[1]], X[msr$inds[4]+1], Y[msr$inds[2]+1])

## End(Not run)
```

mknapsack *Multiple 0-1 Knapsack Problem*

Description

Solves the 0-1 (binary) multiple knapsack problem.

Usage

```
mknapsack(w, p, cap)
```

Arguments

w	vector of (positive) weights.
p	vector of (positive) profits.
cap	vector of capacities of different knapsacks.

Details

Solves the 0-1 multiple knapsack problem for a set of profits and weights.
A multiple 0-1 knapsack problem can be formulated as:

maximize vstar = p(1)*(x(1,1) + ... + x(m,1)) + ... + p(n)*(x(1,n) + ... + x(m,n)) subject to w(1)*x(i,1) + ... + w(n)*x(i,n) <= cap(i) for i=1,...,m x(1,j) + ... + x(m,j) <= 1 for j=1,...,n x(i,j) = 0 or 1 for i=1,...,m , j=1,...,n ,

The multiple knapsack problem is reformulated as a linear program and solved with the help of package lpSolve.

This function can be used for the single knapsack problem as well, but the 'dynamic programming' version in the knapsack function is faster (but: allows only integer values).

The solution found is most often not unique and may not be the most compact one. In the future, we will attempt to 'compactify' through backtracking. The number of backtracks will be returned in list element bs.

Value

A list with components, ksack the knapsack numbers the items are assigned to, value the total value/profit of the solution found, and bs the number of backtracks used.

Note

Contrary to earlier versions, the sequence of profits and weights has been interchanged: first the weights, then profits.

The compiled version was transferred to the knapsack package on R-Forge (see project 'optimist').

References

Kellerer, H., U. Pferschy, and D. Pisinger (2004). Knapsack Problems. Springer-Verlag, Berlin Heidelberg.

Martello, S., and P. Toth (1990). Knapsack Problems: Algorithms and Computer Implementations. John Wiley & Sons, Ltd.

See Also

Other packages implementing knapsack routines.

Examples

```
## Example 1: single knapsack
w <- c( 2,  20, 20, 30, 40, 30, 60, 10)
p <- c(15, 100, 90, 60, 40, 15, 10,  1)
cap <- 102
(is <- mknapsack(w, p, cap))
which(is$ksack == 1)
# [1] 1 2 3 4 6 , capacity 102 and total profit 280

## Example 2: multiple knapsack
w <- c( 40,  60, 30, 40, 20, 5)
p <- c(110, 150, 70, 80, 30, 5)
cap <- c(85, 65)
is <- mknapsack(w, p, cap)
# kps 1: 1,4;  kps 2: 2,6;  value: 345

## Example 3: multiple knapsack
p <- c(78, 35, 89, 36, 94, 75, 74, 79, 80, 16)
w <- c(18,  9, 23, 20, 59, 61, 70, 75, 76, 30)
cap <- c(103, 156)
is <- mknapsack(w, p, cap)
# kps 1: 3,4,5;  kps 2: 1,6,9; value: 452

## Not run:
```

```
# How to Cut Your Planks with R
# R-bloggers, Rasmus Baath, 2016-06-12
#
# This is application of multiple knapsacks to cutting planks into pieces.

planks_we_have <- c(120, 137, 220, 420, 480)
planks_we_want <- c(19, 19, 19, 19, 79, 79, 79, 103, 103,
                    103, 135, 135, 135, 135, 160)
s <- mknapsack(planks_we_want, planks_we_want + 1, planks_we_have)
s$ksack
##  [1] 5 5 5 5 3 5 5 4 1 5 4 5 3 2 4

# Solution w/o backtracking
# bin 1 :  103                         | Rest:  17
# bin 2 :  135                         | Rest:   2
# bin 3 :   79 +  135                  | Rest:   6
# bin 4 :  103 +  135 + 160            | Rest:  22
# bin 5 : 4*19 + 2*79 + 103 + 135      | Rest:   8
#
# Solution with reversing the bins (bigger ones first)
# bin 1 :  103                         | Rest:   4
# bin 2 :  2*19 +    79                | Rest:  20
# bin 3 :   79  +   135                | Rest:   6
# bin 4 : 2*19  +    79 + 135 + 160    | Rest:   8
# bin 5 : 2*103 + 2*135                | Rest:  17
#
# Solution with backtracking (compactification)
# sol = c(1, 4, 4, 1, 1, 3, 4, 5, 5, 5, 5, 4, 2, 3, 4)
# bin 1 : 2*19 +   79                  | Rest:   3
# bin 2 :  135                         | Rest:   2
# bin 3 :   79 +  135                  | Rest:   6
# bin 4 : 2*19 +   79 + 135 + 160      | Rest:   8
# bin 5 : 3*103 + 135                  | Rest:  36

## End(Not run)
```

neldermead *Nelder-Mead Minimization Method*

Description

An implementation of the Nelder-Mead algorithm for derivative-free optimization / function minimization.

Usage

```
neldermead( fn, x0, ..., adapt = TRUE,
             tol = 1e-10, maxfeval = 10000,
   step = rep(1.0, length(x0)))
```

```
neldermeadb(fn, x0, ..., lower, upper, adapt = TRUE,
            tol = 1e-10, maxfeval = 10000,
            step = rep(1, length(x0)))
```

Arguments

`fn`	nonlinear function to be minimized.
`x0`	starting point for the iteration.
`adapt`	logical; adapt to parameter dimension.
`tol`	terminating limit for the variance of function values; can be made *very* small, like `tol=1e-50`.
`maxfeval`	maximum number of function evaluations.
`step`	size and shape of initial simplex; relative magnitudes of its elements should reflect the units of the variables.
`...`	additional arguments to be passed to the function.
`lower`, `upper`	lower and upper bounds.

Details

Also called a ‘simplex’ method for finding the local minimum of a function of several variables. The method is a pattern search that compares function values at the vertices of the simplex. The process generates a sequence of simplices with ever reducing sizes.

The simplex function minimisation procedure due to Nelder and Mead (1965), as implemented by O’Neill (1971), with subsequent comments by Chambers and Ertel 1974, Benyon 1976, and Hill 1978. For another elaborate implementation of Nelder-Mead in R based on Matlab code by Kelley see package ‘dfoptim’.

`eldermead` can be used up to 20 dimensions (then ‘tol’ and ‘maxfeval’ need to be increased). With `adapt=TRUE` it applies adaptive coefficients for the simplicial search, depending on the problem dimension – see Fuchang and Lixing (2012). This approach especially reduces the number of function calls.

With upper and/or lower bounds, `neldermeadb` applies `transfinite` to define the function on all of R^n and to retransform the solution to the bounded domain. Of course, if the optimum is near to the boundary, results will not be as accurate as when the minimum is in the interior.

Value

List with following components:

`xmin`	minimum solution found.
`fmin`	value of `f` at minimum.
`fcount`	number of iterations performed.
`restarts`	number of restarts.
`errmess`	error message

Note

Original FORTRAN77 version by R O'Neill; MATLAB version by John Burkardt under LGPL license. Re-implemented in R by Hans W. Borchers.

References

Nelder, J., and R. Mead (1965). A simplex method for function minimization. Computer Journal, Volume 7, pp. 308-313.

O'Neill, R. (1971). Algorithm AS 47: Function Minimization Using a Simplex Procedure. Applied Statistics, Volume 20(3), pp. 338-345.

J. C. Lagarias et al. (1998). Convergence properties of the Nelder-Mead simplex method in low dimensions. SIAM Journal for Optimization, Vol. 9, No. 1, pp 112-147.

Fuchang Gao and Lixing Han (2012). Implementing the Nelder-Mead simplex algorithm with adaptive parameters. Computational Optimization and Applications, Vol. 51, No. 1, pp. 259-277.

See Also

`hookejeeves`

Examples

```
##  Classical tests as in the article by Nelder and Mead
# Rosenbrock's parabolic valley
rpv <- function(x) 100*(x[2] - x[1]^2)^2 + (1 - x[1])^2
x0 <- c(-2, 1)
neldermead(rpv, x0)                     #  1 1

# Fletcher and Powell's helic valley
fphv <- function(x)
    100*(x[3] - 10*atan2(x[2], x[1])/(2*pi))^2 +
        (sqrt(x[1]^2 + x[2]^2) - 1)^2 +x[3]^2
x0 <- c(-1, 0, 0)
neldermead(fphv, x0)                    #  1 0 0

# Powell's Singular Function (PSF)
psf <- function(x)  (x[1] + 10*x[2])^2 + 5*(x[3] - x[4])^2 +
                    (x[2] - 2*x[3])^4 + 10*(x[1] - x[4])^4
x0 <- c(3, -1, 0, 1)
neldermead(psf, x0)         #  0 0 0 0, needs maximum number of function calls

# Bounded version of Nelder-Mead
lower <- c(-Inf, 0,   0)
upper <- c( Inf, 0.5, 1)
x0 <- c(0, 0.1, 0.1)
neldermeadb(fnRosenbrock, c(0, 0.1, 0.1), lower = lower, upper = upper)
# $xmin = c(0.7085595, 0.5000000, 0.2500000)
# $fmin = 0.3353605

## Not run:
# Can run Rosenbrock's function in 30 dimensions in one and a half minutes:
```

```
neldermead(fnRosenbrock, rep(0, 30), tol=1e-20, maxfeval=10^7)
# $xmin
#  [1]  0.9999998 1.0000004 1.0000000 1.0000001 1.0000000 1.0000001
#  [7]  1.0000002 1.0000001 0.9999997 0.9999999 0.9999997 1.0000000
# [13]  0.9999999 0.9999994 0.9999998 0.9999999 0.9999999 0.9999999
# [19]  0.9999999 1.0000001 0.9999998 1.0000000 1.0000003 0.9999999
# [25]  1.0000000 0.9999996 0.9999995 0.9999990 0.9999973 0.9999947
# $fmin
# [1] 5.617352e-10
# $fcount
# [1] 1426085
# elapsed time is 96.008000 seconds
## End(Not run)
```

`occurs` *Finding Subsequences*

Description

Counts items, or finds subsequences of (integer) sequences.

Usage

```
count(x, sorted = TRUE)

occurs(subseq, series)
```

Arguments

`x`	array of items, i.e. numbers or characters.
`sorted`	logical; default is to sort items beforehand.
`subseq`	vector of integers.
`series`	vector of integers.

Details

`count` counts the items, similar to `table`, but as fast and a more tractable output. If `sorted` then the total number per item will be counted, else per repetition.

If `m` and `n` are the lengths of `s` and `S` resp., then `occurs(s, S)` determines all positions `i` such that `s` `==` `S[i, ..., i+m-1]`.

The code is vectorized and relatively fast. It is intended to complement this with an implementation of Rabin-Karp, and possibly Knuth-Morris-Pratt and Boyer-Moore algorithms.

Value

`count` returns a list with components `v` the items and `e` the number of times it apears in the array. `occurs` returns a vector of indices, the positions where the subsequence appears in the series.

Examples

```
## Examples
patrn <- c(1,2,3,4)
exmpl <- c(3,3,4,2,3,1,2,3,4,8,8,23,1,2,3,4,4,34,4,3,2,1,1,2,3,4)
occurs(patrn, exmpl)
## [1]  6 13 23

## Not run:
set.seed(2437)
p <- sample(1:20, 1000000, replace=TRUE)
system.time(i <- occurs(c(1,2,3,4,5), p))  #=>  [1] 799536
##  user  system elapsed
## 0.017   0.000   0.017 [sec]

system.time(c <- count(p))
##  user  system elapsed
## 0.075   0.000   0.076
print(c)
## $v
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
## $e
##  [1] 49904 50216 49913 50154 49967 50045 49747 49883 49851 49893
## [11] 50193 50024 49946 49828 50319 50279 50019 49990 49839 49990

## End(Not run)
```

setcover *Set cover problem*

Description

Solves the Set Cover problem as an integer linear program.

Usage

```
setcover(Sets, weights)
```

Arguments

`Sets`	matrix of 0s and 1s, each line defining a subset.
`weights`	numerical weights for each subset.

Details

The Set Cover problems attempts to find in subsets (of a 'universe') a minimal set of subsets that still covers the whole set.

Each line of the matrix `Sets` defines a characteristic function of a subset. It is required that each element of the universe is contained in at least one of these subsets.

The problem is treated as an Integer Linear Program (ILP) and solved with the `lp` solver in `lpSolve`.

Value

Returns a list with components `sets`, giving the indices of subsets, and `objective`, the sum of weights of subsets present in the solution.

References

See the Wikipedia article on the "set cover problem".

See Also

knapsack

Examples

```
# Define 12 subsets of universe {1, ..., 10}.
set.seed(7*11*13)
A <- matrix(sample(c(0,1), prob = c(0.8,0.2), size = 120, replace =TRUE),
            nrow = 12, ncol = 10)
sol <- setcover(Sets = A, weights = rep(1, 12))
sol
## $sets
## [1]  1  2  9 12
## $no.sets
##[1] 4

# all universe elements are covered:
colSums(A[sol$sets, ])
## [1] 1 1 2 1 1 1 2 1 1 2
```

`SIAM test functions` *Trefethen and Wagon Test Functions*

Description

Test functions for global optimization posed for the SIAM 100-digit challenge in 2002 by Nick Trefethen, Oxford University, UK.

Usage

```
fnTrefethen(p2)
fnWagon(p3)
```

Arguments

`p2`	Numerical vector of length 2.
`p3`	Numerical vector of length 3.

Details

These are highly nonlinear and oscillating functions in two and three dimensions with thousands of local mimima inside the unit square resp. cube (i.e., [-1, 1] x [-1, 1] or [-1, 1] x [-1, 1] x [-1, 1]).

Value

Function value is a single real number.

Author(s)

HwB <hwborchers@googlemail.com>

References

F. Bornemann, D. Laurie, S. Wagon, and J. Waldvogel (2004). The SIAM 100-Digit Challenge: A Study in High-Accuracy Numerical Computing. Society for Industrial and Applied Mathematics.

Examples

```
  x <- 2*runif(5) - 1
  fnTrefethen(x)
  fnWagon(x)

  ## Not run:
  T <- matrix(NA, nrow=1001, ncol=1001)
  for (i in 1:1001) {
    for (j in 1:1001) {
        T[i, j] <- fnTrefethen(c(x[i], y[j]))
    }
  }
  image(x, y, T)
  contour(x, y, T, add=TRUE)

## End(Not run)
```

`simpleDE` *Simple Differential Evolution Algorithm*

Description

Simple Differential Evolution for Minimization.

Usage

```
simpleDE(fun, lower, upper, N = 64, nmax = 256, r = 0.4,
             confined = TRUE, log = FALSE)
```

Arguments

`fun`	the objective function to be minimized.
`lower`	vector of lower bounds for all coordinates.
`upper`	vector of upper bounds for all coordinates.
`N`	population size.
`nmax`	bound on the number of generations.
`r`	amplification factor.
`confined`	logical; stay confined within bounds.
`log`	logical; shall a trace be printed.

Details

Evolutionary search to minimize a function: For points in the current generation, children are formed by taking a linear combination of parents, i.e., each member of the next generation has the form

$$p_1 + r(p_2 - p_3)$$

where the p_i are members of the current generation and r is an amplification factor.

Value

List with the following components:

`fmin`	function value at the minimum found.
`xmin`	numeric vector representing the minimum.
`nfeval`	number of function calls.

Note

Original Mathematica version by Dirk Laurie in the SIAM textbook. Translated to R by Hans W Borchers.

Author(s)

HwB <hwborchers@googlemail.com>

References

Dirk Laurie. “A Complex Optimization". Chapter 5 In: F. Bornemann, D. Laurie, S. Wagon, and J. Waldvogel (Eds.). The SIAM 100-Digit Challenge. Society of Industrial and Applied Mathematics, 2004.

See Also

`simpleEA`, `DEoptim` in the ‘DEoptim’ package.

Examples

```
simpleDE(fnTrefethen, lower = c(-1,-1), upper = c(1,1))
# $fmin
# [1] -3.306869
# $xmin
# [1] -0.02440308  0.21061243  # this is the true global optimum!
```

---

simpleEA *Simple Evolutionary Algorithm*

Description

Simple Evolutionary Algorithm for Minimization.

Usage

```
simpleEA(fn, lower, upper, N = 100, ..., con = 0.1, new = 0.05,
         tol = 1e-10, eps = 1e-07, scl = 1/2, confined = FALSE, log = FALSE)
```

Arguments

`fn`	the objective function to be minimized.
`lower`	vector of lower bounds for all coordinates.
`upper`	vector of upper bounds for all coordinates.
`N`	number of children per parent.
`...`	additional parameters to be passed to the function.
`con`	percentage of individuals concentrating to the best parents.
`new`	percentage of new individuals not focussed on existing parents.
`tol`	tolerance; if in the last three loops no better individuals were found up to this tolerance, stop.
`eps`	grid size bound to be reached.
`scl`	scaling factor for shrinking the grid.
`confined`	logical; shall the set of individuals be strictly respect the boundary? Default: FALSE.
`log`	logical, should best solution found be printed per step.

Details

Evolutionary search to minimize a function: For each point in the current generation, *n* random points are introduced and the *n* best results of each generation (and its parents) are used to form the next generation.

The scale shrinks the generation of new points as the algorithm proceeds. It is possible for some children to lie outside the given rectangle, and therefore the final result may lie outside the unit rectangle well. (TO DO: Make this an option.)

Value

List with the following components:

`par`	numeric vector representing the minimum found.
`val`	function value at the minimum found.
`fun.calls`	number of function calls made.
`rel.scl`	last scaling factor indicating grid size in last step.
`rel.tol`	relative tolerance within the last three minima found.

Note

Original Mathematica Version by Stan Wagon in the SIAM textbook. Translated to R by Hans W Borchers.

Author(s)

HwB <hwborchers@googlemail.com>

References

Stan Wagon. "Think Globally, Act Locally". Chapter 4 In: F. Bornemann, D. Laurie, S. Wagon, and J. Waldvogel (Eds.). The SIAM 100-Digit Challenge. Society of Industrial and Applied Mathematics, 2004.

See Also

`DEoptim` in the 'DEoptim' package.

Examples

```
simpleEA(fnTrefethen, lower=c(-1,-1), upper=c(1,1), log=FALSE)
# $par
# [1] -0.02440310  0.21061243  # this is the true global optimum!
# $val
# [1] -3.306869
```

`subsetsum` *Subset Sum Problem*

Description

Subset sum routine for positive integers.

Usage

```
subsetsum(S, t, method = "greedy")

sss_test(S, t)
```

Arguments

`S`	vector of positive integers.
`t`	target value, bigger than all items in `S`.
`method`	can be “greedy” or “dynamic”, where “dynamic” stands for the dynamic programming approach.

Details

`subsetsum` is searching for a set of elements in `S` that sum up to `t` by continuously adding more elements of `S`.

It is not required that `S` is decreasingly sorted. But for reasons of efficiency and smaller execution times it is urgently recommended to sort the item set in decreasing order. See the examples to find out how to handle your data.

The first components will be preferred, i.e., if `S` is decreasing, the sum with larger elements will be found, if increasing, the sum with smaller elements. Because of timing considerations, the default is to sort decreasingly before processing.

The dynamic method may be faster for large sets, but will also require much more memory if the target value is large.

`sss_test` will find the biggest number below or equal to `t` that can be expressed as a sum of items in `S`. It will not return any indices. It can be quite fast, though it preprocesses the set `S` to be sorted decreasingly, too.

Value

List with the target value, if reached, and vector of indices of elements in `S` that sum up to `t`.

If no solution is found, the dynamic method will return indices for the largest value below the target, the greedy method witll return NULL.

`sss_test` will simply return maximum sum value found.

Note

A compiled version – and much faster, in Fortran – can be found in package ’knapsack’ (R-Forge, project ’optimist’) as `subsetsum`. A recursive version, returning *all* solutions, is much too slow in R, but is possible in Julia and can be asked from the author.

Author(s)

HwB email: <hwborchers@googlemail.com>

References

Horowitz, E., and S. Sahni (1978). Fundamentals of Computer Algorithms. Computer Science Press, Rockville, ML.

See Also

`maxsub`

Examples

```
t <- 5842
S <- c(267, 493, 869, 961, 1000, 1153, 1246, 1598, 1766, 1922)

# S is not decreasingly sorted, so ...
o  <- order(S, decreasing = TRUE)
So <- S[o]                          # So is decreasingly sorted

sol <- subsetsum(So, t)             # $inds:  2 4 6 7 8  w.r.t.  So
is  <- o[sol$inds]                  # is:     9 7 5 4 3  w.r.t.  S
sum(S[is])                          # 5842

## Not run:
amount <- 4748652
products <-
c(30500,30500,30500,30500,42000,42000,42000,42000,
  42000,42000,42000,42000,42000,42000,71040,90900,
  76950,35100,71190,53730,456000,70740,70740,533600,
  83800,59500,27465,28000,28000,28000,28000,28000,
  26140,49600,77000,123289,27000,27000,27000,27000,
  27000,27000,80000,33000,33000,55000,77382,48048,
  51186,40000,35000,21716,63051,15025,15025,15025,
  15025,800000,1110000,59700,25908,829350,1198000,1031655)

# prepare set
prods <- products[products <= amount]  # no elements > amount
prods <- sort(prods, decreasing=TRUE)  # decreasing order

# now find one solution
system.time(is <- subsetsum(prods, amount))
#  user  system elapsed
# 0.030   0.000   0.029

prods[is]
#  [1]   70740   70740   71190   76950   77382   80000   83800
#  [8]   90900  456000  533600  829350 1110000 1198000

sum(prods[is]) == amount
# [1] TRUE

# Timings:
#             unsorted   decr.sorted
# "greedy"      22.930         0.030    (therefore the default settings)
# "dynamic"      2.515         0.860    (overhead for smaller sets)
# sss_test       8.450         0.040    (no indices returned)

## End(Not run)
```

Testfunctions   *Optimization Test Functions*

Description

Simple and often used test function defined in higher dimensions and with analytical gradients, especially suited for performance tests. Analytical gradients, where existing, are provided with the `gr` prefix. The dimension is determined by the length of the input vector.

Usage

```
fnRosenbrock(x)
grRosenbrock(x)
fnRastrigin(x)
grRastrigin(x)
fnNesterov(x)
grNesterov(x)
fnNesterov1(x)
fnNesterov2(x)
fnHald(x)
grHald(x)
fnShor(x)
grShor(x)
```

Arguments

`x`	numeric vector of a certain length.

Details

Rosenbrock – Rosenbrock's famous valley function from 1960. It can also be regarded as a least-squares problem:

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

No. of Vars.:	n >= 2
Bounds:	-5.12 <= xi <= 5.12
Local minima:	at f(-1, 1, . . . , 1) for n >= 4
Minimum:	0.0
Solution:	xi = 1, i = 1:n

Nesterov – Nesterov's smooth adaptation of Rosenbrock, based on the idea of Chebyshev polynomials. This function is even more difficult to optimize than Rosenbrock's:

$$(1-x_1)^2/4 + \sum_{i=1}^{n-1}(1+x_{i+1}-2x_i^2)^2$$

Two nonsmooth Nesterov functions are available: Nesterov2 and Nesterov1, defined as

$$(1-x_1)^2/4 + \sum_{i=1}^{n-1}|1+x_{i+1}-2x_i^2|$$

$$|1 - x_1|/4 + \sum_{i=1}^{n-1}(|1 + x_{i+1} - 2|x_i||$$

No. of Vars.:	n >= 2
Bounds:	-5.12 <= xi <= 5.12
Local minima: ?	
Minimum:	0.0
Solution:	xi = 1, i = 1:n

Nesterov1 and **Nesterov2** – Simlar to `Nesterov`, except the terms added are taken with absolute value, which makes this function nonsmooth and painful for gradient-based optimization routines; no gradient provided.
(Nesterov2 uses absolute instead of quadratic terms.)

Rastrigin – Rastrigin's function is a famous, non-convex example from 1989 for global optimization. It is a typical example of a multimodal function with many local minima:

$$10n + \sum_{1}^{n}(x_i^2 - 10\cos(2\pi x_i))$$

No. of Vars.:	n >= 2
Bounds:	-5.12 <= xi <= 5.12
Local minima:	many
Minimum:	0.0
Solution:	xi = 0, i = 1:n

Hald – Hald's function is a typical example of a non-smooth test function, from Hald and Madsen in 1981.

$$\max_{1 \leq i \leq n} |\frac{x_1 + x_2 t_i}{1 + x_3 t_i + x_4 t_i^2 + x_5 t_i^3} - \exp(t_i)|$$

where $n = 21$ and $t_i = -1 + (i-1)/10$ for $1 \leq i \leq 21$.

No. of Vars.:	n =5
Bounds:	-1 <= xi <= 1
Local minima:	?
Minimum:	0.0001223713
Solution:	(0.99987763, 0.25358844, -0.74660757, 0.24520150, -0.03749029)

Shor – Shor's function is another typical example of a non-smooth test function, a benchmark for Shor's R-algorithm.

Value

Returns the values of the test function resp. its gradient at that point. If an analytical gradient is not available, a function computing the gradient numerically will be provided.

References

Search the Internet.

Examples

```
x <- runif(5)
fnHald(x); grHald(x)

# Compare analytical and numerical gradient
shor_gr <- function(x) adagio:::ns.grad(fnShor, x)    # internal gradient
grShor(x); shor_gr(x)
```

`transfinite` *Boxed Region Transformation*

Description

Transformation of a box/bound constrained region to an unconstrained one.

Usage

```
transfinite(lower, upper, n = length(lower))
```

Arguments

`lower, upper`	lower and upper box/bound constraints.
`n`	length of upper, lower if both are scalars, to which they get repeated.

Details

Transforms a constraint region in n-dimensional space bijectively to the unconstrained R^n space, applying a `atanh` resp. `exp` transformation to each single variable that is bound constraint.

It provides two functions, `h: B = []x...x[] --> R^n` and its inverse `hinv`. These functions can, for example, be used to add box/bound constraints to a constrained optimization problem that is to be solved with a (nonlinear) solver not allowing constraints.

Value

Returns to functions as components `h` and `hinv` of a list.

Note

Based on an idea of Ravi Varadhan, intrinsically used in his implementation of Nelder-Mead in the ‘dfoptim’ package.

For positivity constraints, `x>=0`, this approach is considered to be numerically more stable than `x-->exp(x)` or `x-->x^2`.

Examples

```
lower <- c(-Inf, 0,   0)
upper <- c( Inf, 0.5, 1)
Tf <- transfinite(lower, upper)
h <- Tf$h; hinv <- Tf$hinv

## Not run:
##  Solve Rosenbrock with one variable restricted
rosen <- function(x) {
    n <- length(x)
    x1 <- x[2:n]; x2 <- x[1:(n-1)]
    sum(100*(x1-x2^2)^2 + (1-x2)^2)
}
f  <- function(x) rosen(hinv(x))    # f must be defined on all of R^n
x0 <- c(0.1, 0.1, 0.1)              # starting point not on the boundary!
nm <- nelder_mead(h(x0), f)         # unconstraint Nelder-Mead
hinv(nm$xmin); nm$fmin              # box/bound constraint solution
# [1] 0.7085596 0.5000000 0.2500004
# [1] 0.3353605

## End(Not run)
```

Index