Portfolio Probe Reference Manual

June 22, 2012

Topics documented:

`build.constraints` *Build Constraints for Optimization*

Description

Builds a bounds object suitable for the input linear constraints, and ensures that the constraints are in a suitable form.

Usage

```
build.constraints(x, bounds = NULL)
```

Arguments

`x`	required. Vector, matrix, data frame or list giving the information about the constraint(s) for each asset. See the Details section for more.
`bounds`	optional matrix of bounds. The rows correspond to the constraints. Rows from this matrix that match constraint names that are created for `x` are put into the output `bounds` matrix.

Value

a list with the following components:

`lin.constraints`
: a matrix or data frame similar to the input `x`. This always has column names so that the constraints can be identified.

`bounds`
: a matrix with two columns and as many rows as constraints represented by the input `x`. Numeric columns of `x` embody one constraint. Columns that are factors or character or logical represent as many constraints as there are unique values in the column.

Details

Numeric linear constraints are created with numeric data. Categorical linear constraints (and count constraints) are created with a factor, character data or logical data.

A data frame is required as the input `x` if more than one type of data is given.

A list of named vectors may be given. The advantage of this is that the vectors need not be in the same order or have the exact same set of assets. The function will return an object for the assets that are in all of the components of the list.

Revision

This help was last revised 2012 June 10.

See Also

`constraints.realized`, `trade.optimizer`,

Examples

```
my.sc <- matrix(c("energy", "telecom", "energy", "Germany", "France",
        "France"), nrow=3, ncol=2, dimnames=list(c("Asset 1", "Asset 2",
        "Asset 3"), c("Sector", "Country")))
my.constr <- build.constraints(my.sc)
my.constr$bounds[,1] <- c(-.05, -.10, 0, .05)
my.constr$bounds[,2] <- c(.15, .15, .20, .55)

random.portfolio(100, prices,
        lin.constraints=my.constr$lin.constraints,
        lin.bounds=my.constr$bounds,
        gross.value=1e6, net.value=c(-1e5, 6e5))

my.constr2 <- build.constraints(list(NumCon=c(A=23, B=72, C=12),
        Country=c(C='Estonia', B='Latvia', D='Estonia')))
```

Cfrag.list *Write a Fragment of a C Program*

Description

Writes either the C declaration of items in a list, or initializes them with the contents of the list.

Usage

```
Cfrag.list(x, file = NULL,  item.num = c(3, 10, 5),
        indent = c("\t", "\t\t"), declaration.only = FALSE,
        long = FALSE, append = FALSE)
```

Arguments

`x`	required. A list with names.
`file`	either `NULL` or a character string. If `NULL` (or an empty string), then a vector of characters is returned.
`item.num`	length three vector giving the number of items per line for doubles, integers and characters.
`indent`	length two vector giving the amount to indent declarations and the items in the initialization.
`declaration.only`	logical flag. If `TRUE`, then the variables are declared but not given initial values.
`long`	logical flag. If `TRUE`, then integers are declared to be "long". If `FALSE`, then integers are declared to be "int".
`append`	logical flag. If `TRUE`, then the file is appended to; otherwise it is overwritten if it exists.

Value

if `file` is a non-empty character string, then the name of the file that is written. Otherwise, a character vector of the declarations – each element representing a different line.

Side Effects

if `file` is a non-empty character string, then the file is created, overwritten or appended.

Details

The type to declare is dependent on the storage mode of the component of `x`. You may need to coerce components to get them to be declared the correct type.

Revision

This help was last revised 2009 September 21.

See Also

`.C`, `storage.mode`, `as.double`, `as.integer`, `as.character`, `cat`.

Examples

```
test.list <- list(adoub=as.double(rnorm(20)), anint=as.integer(92:109),
        achar=c("aaa", "bbbb", "ccccc"))

Cfrag.list(test.list, file="test.c")

Cfrag.list(test.list[1], file="test.c", dec=TRUE)
Cfrag.list(test.list[-1], file="test.c", dec=FALSE, append=TRUE)
```

`constraints.realized` *Show Violations of Linear Constraints*

Description

Returns a list of matrices containing the bounds, the achieved level, plus the amount it violates the nearest bound or else the proximity to the nearest bound if it is not violated. Components of the list correspond to linear (and count) constraints, and distance constraints.

Usage

```
constraints.realized(portfol, lin.constraints, prices = portfol$prices,
        lin.table=portfol$lin.table, lin.trade = NULL, lin.abs = NULL,
        lin.style = NULL, lin.direction = NULL, lin.riskfrac.col = NULL,
        risk.fraction = portfol$risk.fraction, exclude.inf = FALSE)
```

Arguments

`portfol`	required. An object of class `portfolBurSt` (or a similar object).
`lin.constraints`	required. matrix or data frame describing the linear constraints.
`prices`	named vector of prices containing (at least) all of the assets involved with the constraints.
`lin.bounds`	a two-column matrix giving the bounds for (some of) the constraints.

`lin.table` a data frame describing (some of) the constraints. Values within this can be overridden by the other arguments.

`lin.trade` logical vector stating which columns of `lin.constraints` refer to constraints on the trade (as opposed to on the portfolio). This is replicated if necessary.

`lin.abs` logical vector stating which columns of `lin.constraints` are absolute constraints (constraints on the gross as opposed to the net). This is replicated if necessary.

`lin.style` character vector stating the style of each of the linear constraints.

`lin.direction` numeric vector stating the direction of each of the linear constraints.

`lin.riskfrac.col` numeric vector stating the column of `risk.fraction` to be used in the constraint.

`risk.fraction` matrix giving the risk fraction values.

`exclude.inf` logical value stating whether or not (sub)constraints whose bounds are both infinite should be dropped. If all bounds are infinite, this is effectively forced to be `FALSE`.

Value

a list with component `linear` which itself has components:

`constraints` matrix containing the realized value for each (sub)constraint, along with possible information pertaining to bounds.

`lin.table` data frame giving the settings for the constraints.

Details

If there are no bounds given explicitly or by default for the linear constraints, then infinite bounds are created for all of the constraints and `exclude.inf` is forced to be `FALSE`.

This function produces the `con.realized` component of the output of `trade.optimizer` – this component is part of the output of the `summary` for such objects.

Revision

This help was last revised 2012 June 06.

See Also

`build.constraints`, `summary.portfolBurSt`, `trade.optimizer`.

Examples

```
my.sc <- matrix(c("energy", "telecom", "energy", "Germany", "France",
        "France"), nrow=3, ncol=2, dimnames=list(c("Asset 1", "Asset 2",
        "Asset 3"), c("Sector", "Country")))
my.constr <- build.constraints(my.sc)
my.constr$bounds[,1] <- c(-.05, -.10, 0, .05)
my.constr$bounds[,2] <- c(.15, .15, .20, .55)

op <- trade.optimizer(prices, varian,
        lin.constraints=my.constr$lin.constraints,
```

```
        lin.bounds=my.constr$bounds, gross.value=1e6,
        net.value=c(-1e5, 6e5))

constraints.realized(op, my.constr$lin.constraints,
        bounds=my.constr$bounds)

op2 <- trade.optimizer(prices, varian, gross.value=1e6,
        net.value=c(-1e5, 6e5))

# view values for unconstrained portfolio
constraints.realized(op2, my.constr$lin.constraints)
```

`deport.portfolBurSt` *Write Optimization Results to a File*

Description

Creates either a csv (comma separated) file or a txt (tab separated) file that contains the optimized portfolio, the trade, the existing portfolio, or all three.

Usage

```
deport.portfolBurSt(x, filename = deparse(substitute(x)), what = "all",
        multiplier = 1, to = "csv", blank = "")
```

Arguments

`x`	required. An object of class `"portfolBurSt"` (which is an optimized portfolio created by `trade.optimizer`).
`filename`	a character string giving the name of the file to be created, or the empty string (`""`). The extension ("csv" or "txt") need not be a part of the string.
`what`	a character string partially matching one of the following: `"all"`, `"trade"`, `"optimal"`, `"new.portfolio"`, `"existing"`.
`multiplier`	either a single number, or a named vector where the names match the assets that are involved. This is used, for example, when the optimization was performed in terms of lots, but the file is desired to be in shares. Another use is to produce monetary values in the file.
`to`	a character string partially matching one of: `"csv"`, `"txt"`. This controls whether the file is comma-separated or tab-separated.
`blank`	a character string giving the value to be placed in locations that do not occur (in the case when `what = "all"`). The default is no character at all – the other logical choice for this is `"0"`.

Value

a character string giving the name of the file created; or `NULL` if `filename` is the empty string.

Side Effects

the file of the specified name is created or overwritten. Except when `filename` is the empty string, then the values are printed.

Details

The "optimal" and "new.portfolio" choices for the what argument are the same thing.

Revision

This help was last revised 2012 April 17.

See Also

trade.optimizer, write.table.

Examples

```
optim1 <- trade.optimizer(eq.prices, varian, long.only=TRUE,
        gross.value=1e6, ntrade=60, existing=cur.port)

# create file named optim1.csv
deport(optim1)

# print to screen instead
deport(optim1, filename="")

# optimization in lots (of 100), file in shares
deport(optim1, multiplier=100)

# write monetary values in the file
deport(optim1, multiplier=current.prices)

deport(optim1, what="opt", to="txt")

deport(optim1, file="opt1zero", what="all", blank="0")
```

---

deport.randportBurSt *Write Random Portfolios to a File*

Description

Creates a file of the random portfolios represented by the object given it.

Usage

```
deport.randportBurSt(x, filename = deparse(substitute(x)),
        what = "horizontal.portfolio", multiplier = 1,
        names.assets = NULL, to = "csv", blank = "", append = FALSE,
        subset = NULL)
```

Arguments

x	required. Object of class "randportBurSt" (result of random.portfolio and friends).
filename	a character string giving the name of the resulting file, or the empty string (""). The extension (”csv” or ”txt”) need not be a part of the string.

what	a character string describing the layout of the file. This must partially match either "horizontal.portfolio" or "vertical.portfolio".
multiplier	either a single number, or a numeric vector with names. If a vector, the names should be the names of any assets that may appear in x, and the values give the amount by which to multiply each value in x.
names.assets	a character vector providing the names of all assets to be represented in the file. The default is to use the union of the assets that are in the random portfolios.
to	a character string partially matching one of: "csv", "txt". This controls whether the file is comma-separated or tab-separated.
blank	a character string giving the value to be placed in locations that do not occur. The default is no character at all – the other logical choice for this is "0".
append	logical flag. If FALSE, then the file is created or overwritten. If TRUE (which is only allowed for horizontal portfolios), then the file is appended.
subset	if given, a vector that subscripts the random solutions.

Value

a character string of the name of the file created; or NULL if filename is the empty string.

Side Effects

the file is created, overwritten or appended. Except when filename is the empty string, then the values are printed.

Revision

This help was last revised 2012 April 17.

See Also

random.portfolio.

Examples

```
randport1 <- random.portfolio(1000, prices, varian, long.only=TRUE,
        gross.value = 1e6, bench.constraint=c(spx=.04^2/252),
        lin.constraints=cntrysect.conmat, lin.bounds=cntrysect.bounds)

# create file randport1.csv
deport(randport1)

# print to screen instead
deport(randport1, filename="")

# generation in lots, file in shares
deport(randport1, mult=100)

# write monetary values in file
deport(randport1, mult=prices)

# tab-separated file of vertically oriented portfolios
# all asset names from prices used
deport(randport1, to="txt", what="vert", names=names(prices))
```

`head.randportBurSt` *Portion of a Random Portfolio Object*

Description

Returns the first few random portfolios or the last few random portfolios.

Usage

```
head.randportBurSt(x, n = 6)
tail.randportBurSt(x, n = 6)
```

Arguments

`x`	an object of class `randportBurSt` (often created by `random.portfolio`).
`n`	the number of items to select (this is the argument to the default method). This can be a negative number if the default method that is visible allows it.

Value

an object like the input `x` but shorter (usually).

Details

These are methods of the generic functions `head` and `tail`. The output of these functions retains the class attribute. In contrast, regular subscripting will lose the class and the result will be a plain list.

Revision

This help was last revised 2010 January 02.

See Also

`random.portfolio`, `summary.randportBurSt`.

Examples

```
randport1 <- random.portfolio(100, prices, varian, long.only=TRUE,
        bench.constr=c(spx=.04^2/252), lin.constraints=cntrysect.conmat,
        lin.bounds=cntrysect.bounds, gross.value=1e6)

head(randport1, 4) # first 4 random portfolios
randport1[1:4] # same values but class is lost

tail(randport1)
```

`pprobe.checkinput` *Check Some Input Arguments Remain Unchanged*

Description

This is primarily an internal function and seldom of interest for direct use.

Checks to see that two lists are the same. In practice these are lists of portions of some specific arguments.

Usage

```
pprobe.checkinput(new, old, exclude = NULL, suppress.warning = FALSE)
```

Arguments

`new`	a list of the newly created argument checks.
`old`	a list of the original argument checks.
`exclude`	either `NULL` or a character vector of the names of the arguments that should not be checked.
`suppress.warning`	logical value: if `TRUE`, then there is no warning issued when a difference is found.

Value

a single logical value that is `TRUE` if there were no changes.

Side Effects

If a change is found and `suppress.warning` is `FALSE`, then a warning message is issued.

Details

The `checkinput` components only have a few values in them, so it is possible for values to have changed and the check not to notice.

Revision

This help was last revised 2009 December 29.

Examples

```
opt1 <- update(opt0)
pprobe.checkinput(opt1$checkinput, opt0$checkinput)
# a warning is issued if prices, variance, expected.return
# or existing have changed from the original call
```

pprobe.verify *Cursory Test of Portfolio Probe Installation*

Description

Tests if all the functions exist, and if one particular problem works okay.

Usage

```
pprobe.verify()
```

Value

a logical value, returned invisibly. If all seems well, then the value is `TRUE`. When something is wrong, then an error will have occurred.

Side Effects

information is printed on version numbers.

If the function finds something wrong, an error occurs.

Revision

This help was last revised 2009 November 24.

Examples

```
pprobe.verify() # versions are printed
```

random.portfolio *Generate Random Portfolios*

Description

Returns a list of portfolios that satisfy the constraints imposed, but the utility is ignored.

Usage

```
random.portfolio(number.rand=1, ..., out.trade=FALSE,
        seed=NULL, control=random.portfolio.control, identity=NULL)
```

Arguments

`number.rand`	the number of random portfolios desired. This can be zero (in which case `NULL` is returned), but can not be negative.
`...`	arguments of `trade.optimizer` need to be given that specify the constraints. Arguments concerning utility are ignored.
`out.trade`	logical value stating if it should be the random trades (`TRUE`), or the resulting random portfolios (`FALSE`) that are to be returned. The full name ("out.trade") needs to be given – no abbreviations are allowed.

`seed` either `NULL`, a single integer or a random seed to use for the computations. If `NULL` or an integer, then a random seed is created by calling `seed.BurSt`.

The full name ("seed") needs to be given – no abbreviations are allowed.

`control` a list like the output of `random.portfolio.control` or a function that produces such a list.

The full name ("control") needs to be given – no abbreviations are allowed.

`identity` an arbitrary object, most likely a string or an integer. This is merely added unchanged to the output. It is useful when generation of portfolios is repeatedly done in parallel. The whole name ("identity") must be used – no abbreviations allowed.

Value

a list of the random portfolios or trades. This has attributes:

`call` an image of the call that created the object.

`timestamp` two character strings giving the time and date of creation (the start time and the end time of the call).

`seed` the random seed for the generator internal to the C code.

`version` a vector stating the versions of C and S code used. This attribute is suppressed when the object is printed.

`checkinput` list with components `prices`, `variance`, `expected.return` and `existing` that contains a few values of each input. This is used in `randport.eval` and `update` to make sure that the data for these inputs are what you expect they are. This attribute is suppressed when the object is printed.

`funevals` the number of portfolios that were examined during the search. This attribute is suppressed when the object is printed.

`iterhistory` only present if the `save.iterhistory` control argument is `TRUE`, and is seldom of any interest. If present, it is a numeric vector giving the value of the objective at each iteration of the algorithm for the last random portfolio that is generated. The value of the objective is a measure of how much a given portfolio violates the constraints.

`identity` the input value of `identity`. This is suppressed when the object is printed.

`class` `"randportBurSt"`. There are several methods for this class.

Side Effects

the S language random seed is created or changed (unless the `seed` argument is given explicitly and is not `NULL`).

An error occurs if no portfolios were found that satisfy the constraints, unless `throw.error` (a control variable) is set to `FALSE`. This generally only happens if the constraints are inconsistent, or at least very restrictive. In the latter case, you can adjust control arguments to search more thoroughly.

Details

It is possible that fewer portfolios are generated than requested. This happens if too many attempts fail. There is a warning for this unless the `do.warn` item `randport.fail` is set to `FALSE`.

If you want a bound on the utility, then you can use `random.portfolio.utility`. However, this is substantially slower than using `random.portfolio`.

There are `randportBurSt` methods for several generic functions.

When tracing is done, a message gives: the number of optimizations, the number of satisfactory portfolios found, and the number of failures.

Bugs

Tracing generally does not work under Windows.

Revision

This help was last revised 2012 April 17.

See Also

`trade.optimizer`, `random.portfolio.control`, `random.portfolio.utility`, `randport.eval`, `deport.randportBurSt`, `summary.randportBurSt`, `valuation.randportBurSt`, `head.randportBurSt`, `update.randportBurSt`, `seed.BurSt`.

Examples

```
randport1 <- random.portfolio(100, prices, varian, long.only=TRUE,
        bench.constr=c(spx=.04^2/252), lin.constraints=cntrysect.conmat,
        lin.bounds=cntrysect.bounds, gross.value=1e6)

randtrade1 <- random.portfolio(100, prices, varian, long.only=TRUE,
        bench.constr=c(spx=.04^2/252), lin.constraints=cntrysect.conmat,
        lin.bounds=cntrysect.bounds, gross.value=1e6, out.trade=TRUE)

# now use it
randalpha1 <- numeric(length(randtrade1))
for(i in 1:length(randtrade1)) {
        randalpha1[i] <- trade.optimizer(prices, varian, long.only=TRUE,
          bench.constr=c(spx=.04^2/252), lin.constraints=cntrysect.conmat,
          lin.bounds=cntrysect.bounds, gross.value=1e6, funeval=0,
          start=randtrade1[[i]])$alpha.values
}

# same thing again
randalpha1 <- unlist(randport.eval(randtrade1, keep="alpha.values"))
# and again
randalpha1 <- unlist(randport.eval(randport1, keep="alpha.values"))
```

`random.portfolio.control` *Optimizer Controls for Random Portfolios*

Description

Sets parameters that control the optimization of `random.portfolio`.

Usage

```
random.portfolio.control(iterations.max = 20, miniter = 5,
        fail.iter = 5, gen.fail = 4, init.fail = 4,
        throw.error = TRUE,  lockcon = FALSE,
        enforce.max.weight = TRUE, trace = FALSE,
        save.iterhistory = FALSE, safe.mode = TRUE, ...)
```

Arguments

`iterations.max`
: integer giving the maximum number of iterations to perform in a single try at a portfolio.

`miniter`
: integer giving the minimum number of iterations to be tried in each go even if `fail.iter` says to give up.

`fail.iter`
: integer giving the maximum number of consecutive iterations that fail to improve the solution without the algorithm stopping. For example, if `fail.iter` is 3, then the algorithm continues with 3 consecutive failures but stops upon the 4th consecutive failure.

`gen.fail`
: integer limiting the total number of failures allowed in a call. The maximum number of failures allowed is `gen.fail` times `number.rand`.

`init.fail`
: integer giving the number of failures allowed before the first success at finding a portfolio satisfying all the constraints. If the number of initial failures is violated, then it quits.

`throw.error`
: logical value; if `TRUE`, then failing to create any portfolios that satisfy the constraints will cause an error. If `FALSE`, then no error happens and you can tell that no satisfactory portfolios were found because the length of the result is zero.

`lockcon`
: logical value; if `TRUE`, then the constraint penalties are used as given. Otherwise the penalties for some constraints may be changed to try to find portfolios more efficiently.

`enforce.max.weight`
: logical value; if `TRUE`, then forced trades are automatically created if any positions in the existing portfolio break their maximum weight constraint. This doesn't absolutely guarantee that the constraints will be met, but they generally will be unless the gross value is given a lot of latitude.

`trace`
: logical value. If `TRUE`, then information on the progress of the search is printed. This is often ignored under Windows.

`save.iterhistory`
: logical value; if `TRUE`, then a vector will be returned containing the value of the objective at each iteration of the algorithm for the final random portfolio created.

`safe.mode`	logical value; if `FALSE` (not recommended), then some errors may be bypassed. The reason for this option is to allow possible workarounds for bugs if they appear.
`...`	additional arguments for compatibility with `trade.optimizer.control` (because the two tasks use the same code internally).

Value

a list with the following components:

`icontrol`	vector of the integer-valued control parameters.
`dcontrol`	vector of the double precision control parameters.
`aux`	vector of the auxiliary control parameters.

Details

ERRORS. Setting the `throw.error` argument to `FALSE` can be useful when doing series of random portfolios, as when imitating a backtest. It can be the case that the desired constraints are infeasible – you can relax some constraints if this proves to be true and try again.

Revision

This help was last revised 2012 April 17.

See Also

`random.portfolio`, `random.portfolio.utility`, `trade.optimizer.control`.

Examples

```
# it is unlikely to be useful to call this function directly
```

`random.portfolio.utility`

Generate Random Portfolios with a Utility Constraint

Description

Returns a list of portfolios that satisfy the constraints imposed, and includes a constraint on the utility.

Usage

```
random.portfolio.utility(number.rand = 1, objective.limit, prices,
        variance = NULL, expected.return = NULL, gen.fail = 4,
        objfail.max = 1, ..., out.trade = FALSE, verbose = FALSE)
```

Arguments

`number.rand`	the number of random portfolios desired. This can be zero (in which case `NULL` is returned), but can not be negative.
`objective.limit`	a number that gives the upper bound for the objective. The objective is the negative of the utility (except for minimum variance but you should use `random.portfolio` for this utility).
`prices`	the vector of prices. See the `trade.optimizer` help file for the description.
`variance`	the variance. See the `trade.optimizer` help file for the description.
`expected.return`	the expected returns. See the `trade.optimizer` help file for the description.
`gen.fail`	a single number. The maximum number of failures that are allowed is `gen.fail` times `number.rand`.
`objfail.max`	a single number – the maximum number of times that a try can fail to achieve the objective limit.
`...`	arguments of `trade.optimizer` need to be given that specify the constraints and the utility.
`out.trade`	logical value stating if it should be the random trades (`TRUE`), or the resulting random portfolios (`FALSE`) that are to be returned. The full name of this argument needs to be given.
`verbose`	a logical value. If `TRUE`, then messages about the progress are printed. The full name of this argument needs to be given.

Value

a list of the random portfolios or trades. This has attributes:

`call`	an image of the call that created the object.
`timestamp`	a character string giving the time and date of creation.
`seed`	the random seed for the generator internal to the C code for the first try.
`version`	a vector stating the versions of C and S code used. This attribute is suppressed when the object is printed.
`checkinput`	a list with components `prices`, `variance`, `expected.return` and `existing` that contains a few values of each input. This is used in `randport.eval` and `update` to make sure that the data for these inputs are what you expect they are. This attribute is suppressed when the object is printed.
`funevals`	the value `NA`. This attribute is suppressed when the object is printed.
`class`	`"randportBurSt"`. There are several methods for this class.

Side Effects

the S language random seed is created or changed.

Details

It is possible that fewer portfolios are generated than requested. There is a warning for this.

There are two types of failure:

1) The try can fail to decrease the objective enough. If this happens on the first try and `objfail.max=1` (the default), then an error occurs. Otherwise, computation stops if the number of such failures is `objfail.max` – the result contains the portfolios or trades that have been successful (which theoretically could be none).

2) The try could achieve the necessary objective but fail on other constraints. This is likely to be a rare event.

Although `random.portfolio.utility` and `random.portfolio` return objects that are essentially the same, their internal operation is quite different. `random.portfolio` calls the underlying C code once and uses a search that is optimized to find portfolios that satisfy constraints. In contrast `random.portfolio.utility` makes a call to `trade.optimizer` on each try. (The starting point is made totally random and it is told to stop once it achieves the objective limit, but is otherwise the same as ordinary calls to `trade.optimizer`.)

The `seed` argument is ignored. If you want to have reproducible results, then you need to set the S language seed.

There are `randportBurSt` methods for several generic functions.

Bugs

For safety, the `start.sol` argument is ruled out. Unfortunately, an error is thrown even if `start.sol` is given but `NULL`.

Revision

This help was last revised 2012 April 17.

See Also

`trade.optimizer`, `random.portfolio`, `randport.eval`, `deport.randportBurSt`, `summary.randportBurSt`, `valuation.randportBurSt`, `head.randportBurSt`, `update.randportBurSt`.

Examples

```
randport1 <- random.portfolio.utility(100, objective.limit=-.2, prices,
        varian, expected.return=alphas, long.only=TRUE,
        bench.constr=c(spx=.04^2/252), lin.constraints=cntrysect.conmat,
        lin.bounds=cntrysect.bounds, gross.value=1e6)

randtrade1 <- random.portfolio.utility(100, objective.limit=-.2, prices,
        varian, expected.return=alphas, long.only=TRUE,
        bench.constr=c(spx=.04^2/252), lin.constraints=cntrysect.conmat,
        lin.bounds=cntrysect.bounds, gross.value=1e6, out.trade=TRUE)

# now use it
randalpha1 <- numeric(length(randtrade1))
for(i in 1:length(randtrade1)) {
        randalpha1[i] <- trade.optimizer(prices, varian, long.only=TRUE,
          bench.constr=c(spx=.04^2/252), lin.constraints=cntrysect.conmat,
          lin.bounds=cntrysect.bounds, gross.value=1e6, funeval=0,
          start=randtrade1[[i]])$alpha.values
```

```
}

# same thing again
randalpha1 <- unlist(randport.eval(randtrade1, keep="alpha.values"))
# and again
randalpha1 <- unlist(randport.eval(randport1, keep="alpha.values"))
```

`randport.eval` *Evaluate Each Random Portfolio*

Description

Takes a random portfolio object and evaluates each individual portfolio. Any set of components of the output of `trade.optimizer` can be saved.

Usage

```
randport.eval(x, keep = c("results", "alpha.values", "var.values",
        "utility.values"), subset = NULL, do.warn = FALSE,
        additional.args = NULL, checkinput = TRUE,
        envir = parent.frame(), FUN = NULL, ...)
```

Arguments

`x`	required. An object that is the result of a call to `random.portfolio` or `random.portfolio.utility`.
`keep`	character vector of component names of the output of `trade.optimizer`. If this is `NULL`, then the slightly ironic result is that all components are kept. Values are allowed to partially match the names of optimization object components.
`subset`	either `NULL` or a vector to subscript along the components of `x`.
`do.warn`	the control for the warnings coming from `trade.optimizer`. The default is `FALSE`, meaning all warnings are turned off that can be turned off. The two other likely values are: `TRUE` (turn on all warnings), and `NULL` (use the `do.warn` value that was in effect to start with). (A value of `NULL` here is not necessarily the same as `NULL` in `trade.optimizer`.) This can be any value that the `do.warn` argument to `trade.optimizer` will accept.
`additional.args`	list of arguments to `trade.optimizer` either not in the `random.portfolio` call or that should be changed. In the latter case, argument names need to be given exactly the same as in the original – that is, with the same amount of abbreviation.
`checkinput`	a logical value. If `TRUE`, then a (fallible) check is made to see if there is a change in the value of `price`, `variance`, `expected.return` or `existing` between the current computation and the original computation. There is no warning about arguments that are in `additional.args`.
`envir`	the environment where the objects used to generate the random portfolios are going to be found. This is nonsensical to S+.
`FUN`	a function or a character string naming a function. Each portfolio trade object will be passed to the function.
`...`	additional arguments given to `FUN`.

Value

a list as long as `x` (or by what is indicated by `subset` if given).

If `FUN` is not given, each component is a list containing the components of the output of `trade.optimizer` listed in `keep`.

If `FUN` is given, then each component is the result of `FUN` applied to the portfolio trade object for the component.

Details

The objects that are involved (like prices and variances) need to be available in the current session, and presumably should be unchanged.

If you get a warning about a "checkinput difference" for one or more arguments, then you probably want to add those arguments to `additional.args`. Problems like this can arise because the arguments for the original computation were dependent on the iteration in a loop. Outside that original loop the value may be interpreted differently (or cause an error).

Bugs

The default value of `out.trade` in `random.portfolio` (and `random.portfolio.utility`) must remain `FALSE` in order for this to work properly.

The `utility.values` component will be wrong (zero) if `expected.return` is not given and `utility` is not specified.

Revision

This help was last revised 2012 April 17.

See Also

`random.portfolio`, `trade.optimizer`, `random.portfolio.utility`.

Examples

```
randport1 <- random.portfolio(100, prices, variance=varian,
        long.only=TRUE, bench.constr=c(spx=.04^2/252),
        lin.constraints=cntrysect.conmat,
        lin.bounds=cntrysect.bounds, gross.value=1e6)

randeval1 <- randport.eval(randport1)

# look at linear constraints of the random portfolios
randlincon1 <- randport.eval(randport1, keep="con.realized")

# returns the above plus more
randsummary <- randport.eval(randport1, FUN=summary)

# vector of weight differences
rdist <- unlist(randport.eval(randport1, FUN = function(x)
    trade.distance(x, some.portf, prices=privec, scale="weight")))
```

`seed.BurSt` *Create Random Seed for Burns Statistics Functions*

Description

Creates a random seed that is used internally in Burns Statistics functions.

Usage

```
seed.BurSt(n = NULL)
```

Arguments

`n`	if this has length, then `set.seed` is called with this as its argument.

Value

a numeric vector that is the proper length and structure for a random seed for the random generator that is internal to several Burns Statistics functions.

Side Effects

The S language random seed `.Random.seed` is created or changed.

Details

The random generator is an implementation of the Mersenne Twister.

See `http://www.math.keio.ac.jp/~matumoto/`

Revision

This help was last revised 2009 November 09.

`summary.portfolBurSt` *Information on Portfolio Trade Objects*

Description

The summary method shows pertinent information about the object. The print method shows most of the components of the object.

Usage

```
summary.portfolBurSt(object, prices = object$prices)
print.portfolBurSt(x, ...)
```

Arguments

`object`	required. An object of class `"portfolBurSt"`, created by `trade.optimizer`.
`x`	required. An object of class `"portfolBurSt"`, created by `trade.optimizer`.
`prices`	a named vector of prices so that the portfolio can be valued.
`...`	additional arguments to `print`.

Value

`summary` returns a list with the following components:

`results` vector giving the objective, the negative utility (including costs), the cost of the trade, and the penalty imposed for broken constraints.

`objective.utility` character string stating the utility being optimized.

`alpha.values` vector of optimal expected returns.

`var.values` vector of optimal variances.

`number.of.assets` a vector giving the size (number of names) of the existing portfolio, the size of the trade, the size of the new portfolio, the number of positions that are opened, the number of positions that are closed, the number of assets in the universe, the number of tradable assets, plus two more elements that might have affected the number of tradable assets. The `select.universe` element says how long the `universe.trade` argument (effectively) was, and the `positions.notrade` element says how many assets the `positions` argument disallows from trading.

`opening.positions` the asset names of the positions that are opened.

`closing.positions` the asset names of the positions that are closed.

`valuation.new.portfolio` the gross, net, long and short values of the new portfolio. This is present only if there are `prices`.

`valuation.trade` the gross, net, long and short values of the trade. This is present only if there are `prices`.

`valuation.trade.fraction.of.gross` the gross, net, long and short values of the trade divided by the gross value of the portfolio. This is present only if there are `prices`.

`violated` vector of character strings stating which types of constraints were violated. This is absent if all constraints were satisfied.

`risk.fraction` a matrix giving the fraction of variance accountable to each asset. This is present only if the `risk.fraction` argument was specified in the optimization.

`con.realized` the realization of constraints. This is present only if linear and/or distance constraints were specified.

`lin.trade` logical vector stating if linear constraints were on the trade or the portfolio. This is present only if linear constraints were specified.

`lin.abs` logical vector stating if linear constraints were on the gross or the net. This is present only if linear constraints were specified.

`lin.style` vector of character strings stating the style of the linear constraints, that is, `"weight"`, `"value"` or `"count"`. This is present only if linear constraints were specified.

`lin.direction` numeric vector stating the direction of the linear constraints. This is present only if linear constraints were specified.

`dist.style` vector of character strings stating the style of the distance constraints, that is, `"weight"`, `"value"`, `"shares"`, etc. This is present only if distance constraints were specified.

`dist.trade` logical vector stating if distance constraints were on the trade or the portfolio. This is present only if distance constraints were specified.

`dist.utility` logical vector stating if the distances are used in the utility or constrained. This is present only if distance constraints were specified.

`print` invisibly returns the input `x`.

Side Effects

`print` causes a list of some but not all of the components of the object to be printed.

Details

These are methods of `summary` and `print` for objects of class `"portfolBurSt"`.

Revision

This help was last revised 2011 March 24.

See Also

`trade.optimizer`, `constraints.realized`, `valuation.portfolBurSt`.

Examples

```
op1 <- trade.optimizer(eq.prices, varian, long.only=TRUE,
       gross.value=1e6)
op1 # print the object

names(op1) # see names of the object, hence what is left out

summary(op1)
summary(op1, price=new.prices)
```

---

summary.randportBurSt

View Characteristics of Random Portfolios

Description

The summary method counts the number of portfolios of each size, and the number of times each asset appears.

Usage

```
summary.randportBurSt(object)
print.randportBurSt(x, ...)
```

Arguments

object	an object of class randportBurSt as created by random.portfolio or random.portfolio.utility.
x	an object of class randportBurSt as created by random.portfolio or random.portfolio.utility.
...	additional arguments to print may be given.

Value

the summary method returns a list with the following components:

port.size	a vector whose names give sizes of portfolios, and whose values give the number of portfolios of that size.
count.assets	a vector stating the number of occurrences of each asset.

the print method returns its input invisibly.

Side Effects

the print method prints the object, including its attributes. It does, however, only print a few elements of the seed attribute, and does not print the funevals, version or checkinput attributes.

Details

These are methods of summary and print for objects of class "randportBurSt".

Revision

Last revised 2010 January 02.

See Also

random.portfolio, random.portfolio.utility, valuation.randportBurSt, head.randportBurSt.

Examples

```
randport1 <- random.portfolio(100, prices, varian, long.only=TRUE,
        bench.constr=c(spx=.04^2/252), lin.constraints=cntrysect.conmat,
        lin.bounds=cntrysect.bounds, gross.value=1e6)

randport1 # prints the random portfolios

summary(randport1)
```

`trade.distance` *Trade distance*

Description

Returns the trade distance between two portfolios.

Usage

```
trade.distance(x, y, prices = NULL, scale = TRUE, tol = 1e-6)
```

Arguments

`x`	an object that is the result of `trade.optimizer` or a named vector representing a portfolio.
`y`	an object that is the result of `trade.optimizer` or a named vector representing a portfolio.
`prices`	a named vector of prices of the assets in the portfolios. If `NULL`, then the `prices` components from `x` and/or `y` are used.
`scale`	either a logical value or a string that is (an abbreviation of) one of: `"x"`, `"y"`, `"min"`, `"max"`, `"mean"`, `"weight"`. If `FALSE`, then the answer is the value (buys plus sells) of the trade. If `TRUE`, then the trade value is divided by the gross value of `x`. If a string, then the trade value is divided by the gross value of one of the portfolios, by the minimum gross value, by the maximum gross value, or by the mean gross value. If `"weight"`, then the weights are found first and the differences are taken afterwards – in this case the distance can be zero if the gross values of the two portfolios differ. Also in this case, `"x"` and/or `"y"` can be given as weight vectors – if the sum of absolute values is within `tol` of 1, then the vector is considered a weight vector.
`tol`	a single number giving the tolerace for testing if a vector is a weight vector, and the relative tolerance for two prices for the same asset being equal.

Value

a number. If scaling is done and the gross values of the two portfolios are equal, then the maximum possible is 2.

Revision

This help was last revised 2010 April 28.

See Also

`trade.optimizer`.

Examples

```
op1 <- trade.optimizer(prices, varmat, gross.value=9e7,
        long.only=TRUE)
op2 <- update(op1, seed=NULL)
trade.distance(op1, op2)
```

`trade.optimizer` *Select an Optimal Trade for a Portfolio*

Description

Finds an approximately optimal portfolio trade. The required inputs are the prices, and a sufficient indication of the monetary size of the portfolio or the turnover. Typically a variance matrix and possibly a vector of expected returns are given as well. Almost always only integer values are traded, thus round lotting is automatically done if the prices are given for lots. Other constraints include the number of assets traded, and the number of assets in the optimal portfolio.

Usage

```
trade.optimizer(prices, variance=NULL, expected.return=NULL,
        existing=NULL, gross.value=NULL, net.value=NULL, long.value=NULL,
        short.value=NULL, turnover=NULL, long.only=FALSE, max.weight=1,
        universe.trade=NULL, lower.trade=NULL, upper.trade=NULL,
        risk.fraction=NULL, rf.style="fraction", rf.loc=NULL, ntrade=NULL,
        port.size=NULL, threshold=NULL, forced.trade=NULL, positions=NULL,
        tol.positions=0, lin.constraints=NULL, lin.bounds=NULL,
        lin.trade=FALSE, lin.abs=TRUE, lin.style="weight", lin.direction=0,
        lin.rfloc=NULL, alpha.constraint=NULL, var.constraint=NULL,
        bench.constraint=NULL, dist.center=NULL, dist.style="weight",
        dist.bounds=NULL, dist.trade=FALSE, dist.utility=FALSE,
        dist.prices=NULL, sum.weight=NULL, limit.cost=NULL, close.number=NULL,
        utility=NULL, risk.aversion=1, benchmark=NULL, bench.trade=FALSE,
        bench.weights=NULL, long.buy.cost=NULL,
        long.sell.cost=long.buy.cost, short.buy.cost=long.buy.cost,
        short.sell.cost=long.buy.cost, cost.par=NULL, ucost=1,
        scale.cost="gross", start.sol=NULL, allowance=.9999, do.warn=NULL,
        penalty.constraint=1000, quantile=0.5, dest.wt=NULL,
        utable=NULL, atable=NULL, vtable=NULL, dumpfile="", ...,
        seed=.standard.seed.BurSt, control=trade.optimizer.control,
        identity=NULL)
```

Arguments

`prices`	required. Named vector of prices of the assets. This vector determines the assets to be used in the problem – other inputs such as `variance` and `expected.return` may contain information on assets in addition to the ones in `prices`. If there is one or more benchmarks, then these assets do not need to be in `prices`. (But if benchmark trading is allowed, the benchmarks do need to be in `prices`.)
`variance`	numeric matrix or three-dimensional array. This may have assets in addition to those named in `prices` – such assets will be ignored (except for benchmarks). If a three-dimensional array, then each variance matrix is in a different slice of the third dimension. Benchmarks do not need to be given in the variance when their weights (of assets in the variance) are given in `bench.weights`.

Alternatively, compact forms of variances may be given via a list with special components (see the User's Manual) – this is very seldom useful.

`expected.return`
: numeric vector or matrix of expected returns of the assets. If a matrix, then each vector of expected returns is in a different column. This need not have (row) names, but it is generally advised. If it does have asset names, then it can contain information for assets in addition to those in `prices`.

`existing`
: named vector of units (shares or lots) of the existing portfolio. The `prices` vector must contain all of the assets in `existing`. If this is the result of a call to `trade.optimizer` – that is, an object of mode `portfolBurSt` – then the `new.portfolio` component is used as the vector. A random portfolio object containing a single portfolio is also allowed. If not given, then it is presumed there is no existing portfolio (only cash).

  `start.sol` (suggested trade for the optimizer) is sometimes confused with `existing` (initial portfolio). Don't confuse them.

`gross.value`
: numeric vector of length one or two giving the desired gross value of the optimal portfolio. If of length two, this is a range of values that are allowed. If of length one, then the lower bound is created by multiplying by `allowance`. There is more on this argument in the Details section.

`net.value`
: numeric vector of length two (or possibly one) giving the range that the net value may have. There is more on this argument in the Details section.

`long.value`
: numeric vector of length one or two giving the desired sum of the values of the long positions of the optimal portfolio. If of length two, this is a range of values that are allowed. If of length one, then the lower bound is created by multiplying by `allowance`. There is more on this argument in the Details section.

`short.value`
: numeric vector of length one or two giving the desired sum of the value of the short positions of the optimal portfolio. If of length two, this is a range of values that are allowed. If of length one, then the lower bound is created by multiplying by `allowance`. (This is meant to be positive numbers, but negative numbers work as well.) There is more on this argument in the Details section.

`turnover`
: numeric vector of length one or two giving the desired sum of the absolute value of all the trades. If of length one, then the maximum turnover; otherwise the range of allowable trade values. There is more on this argument in the Details section.

`long.only`
: logical value stating if the optimal portfolio is restricted to be long-only (no negative numbers of units of assets).

`max.weight`
: a numeric vector containing numbers between zero and one giving the maximum weight allowed in the gross value for the assets. If this is an unnamed vector, then it is replicated to have length equal to the number of assets. If named, then it bounds those specified assets. `NULL` is (grudgingly) allowed to mean no maximum weight constraints.

`universe.trade`
: character vector giving the names of all the assets that are allowed to be traded. Assets not named here will be disallowed from trading. If `NULL`, then this does not impose a limit on the assets that are allowed to trade.

`lower.trade`
: numeric vector of the desired lower bound on the units to trade in the assets. If this is an unnamed vector, then it is replicated to have length

equal to the number of assets. If named, then it bounds just those specified assets. Bounds may automatically be placed on the assets from other information about the problem. See also the `max.weight` argument and the `positions` argument. There is more on this argument in the Details section.

`upper.trade` numeric vector of the desired upper bound on the units to trade in the assets. If this is an unnamed vector, then it is replicated to have length equal to the number of assets. If named, then it bounds just those specified assets. Bounds may automatically be placed on the assets from other information about the problem. See also the `max.weight` argument and the `positions` argument. There is more on this argument in the Details section.

`risk.fraction` a number, a vector, a two-column matrix, a three-dimensional array with two columns, a list, or `NULL`. This specifies constraints on some measure of the variance attributable to each asset. The `rf.style` argument controls what is being constrained.

The maximum allowed is constrained for all assets by a single number, or for named assets in a vector. If a matrix is given, then the first column is the minimum allowed and the second column is the maximum allowed – the rows should be named with asset ids (not all assets need be constrained).

If there are multiple combinations of variances and benchmarks or multiple risk fraction styles are used, then a three-dimensional array can be used to specify constraints for as many of the situations as desired.

An alternative to giving a three-dimensional array is to give a list with length equal to the number of variance-benchmark-style combinations. The big advantage of a list is that simple values, such as a single number, can be given in some or all of the components.

See the `rf.loc` argument.

`rf.style` character vector giving the style of each risk fraction constraint. That is, it might say that some are value constraints rather than fraction constraints. The length of this should either be 1 or the number of risk fraction constraints. The allowable values are (abbreviations of): `"fraction"`, `"value"`, `"marginalbench"`, `"valmargbench"`, `"corport"`, `"abscorport"`, `"incorport"`, `"absincorport"`.

`"fraction"` constrains the fraction of variance attributed to each of the assets. `"value"` constrains the variance attributed to each of the assets. `"marginalbench"` constrains the fraction of variance relative to the marginal impact of the benchmark. `"valmargbench"` constrains the variance relative to the marginal impact of the benchmark.

`"corport"` constrains the correlation of assets to the portfolio. `"abscorport"` constrains the absolute correlation of assets to the portfolio.

`"incorport"` and `"absincorport"` are like `"corport"` and `"abscorport"`, respectively, except that the constraints only apply to assets that are in the portfolio.

`rf.loc` numeric vector giving the zero-based columns in `vtable` that each risk fraction constraint refers to. This should be `NULL` (interpreted as 0), a single number, or of length equal to the number of risk fraction constraints.

`ntrade` an integer vector of length one or two. If length one, an integer giving the maximum number of assets to trade. If length two, then the minimum and

maximum number of assets to trade. if NULL, then there is no constraint on the number of assets traded.

port.size — an integer or a vector of two integers. If a single integer, the upper bound on the desired number of assets in the optimal portfolio. If two integers, then the allowed range of the number of assets in the optimal portfolio.

threshold — a vector (with names); or a matrix with 1, 2, 3 or 4 columns, and row names. The first two columns give the minimum amount to be traded for (some of) the assets if the asset is traded at all. If a vector or one-column matrix, then the constraint is considered to be symmetric for buying and selling. If a two-column matrix, then the minimum of the two numbers for each asset can not be positive, and the maximum of the two numbers can not be negative.

The third and fourth columns give the minimum number of units (shares or lots) allowed in the portfolio if the asset is in the portfolio at all.

A three-column matrix can only be given for long-only portfolios, in which case the third column is the minimum number of units for the assets if the asset is in the portfolio. Otherwise, the third column should be non-positive.

There is no requirement for row names if all of the rows are the same.

forced.trade — a named numeric vector giving a minimum amount to trade for specific assets (which are identified by the names of the vector). If you want to trade an exact amount, you need to also use upper.trade or lower.trade.

positions — a numeric matrix with 2, 4 or 8 columns giving the monetary value of positions that are allowed for each asset (rows). The first two columns give the minimum and maximum amount of money allowed in the portfolio for each of the assets. The third and fourth columns give the minimum and maximum amount of money that can be traded. Forced trades can be implied. If there is at least one NA in the first 4 columns in a row, then that asset is not allowed to trade (and other values in that row are ignored). Columns 5 through 8 are for threshold constraints: columns 5 and 6 give the minimum money amounts to trade the assets, and columns 7 and 8 give the minimum money amounts allowed in the portfolio. The matrix must have row names. The assets need not match the assets in prices – if there is a difference, then there is a warning unless the positions.names element of do.warn is set to FALSE.

tol.positions — a number giving a monetary value of the tolerance for existing positions to break the portfolio constraints given in the positions argument. For example if the value for an asset in the second column of positions is 2000 and the value of the existing position in that asset is 2010, then a sell will be forced if tol.positions is less than 10 but not if it is more than 10.

lin.constraints — a matrix or data frame describing the linear (and/or count) constraints to be placed on the portfolio and/or trade. Factors or characters (which are coerced to factors) represent as many constraints as there are levels in the factor. It is often easiest if this and lin.bounds are the result of a call to build.constraints. This may correspond to assets in addition to those in the current problem, but must contain information for all of the assets that are in the current problem.

lin.bounds — a two-column matrix with rows that correspond to the constraints represented by lin.constraints (not necessarily in the same order). The

values of the bounds are in terms that are controlled by `lin.style`. If `lin.constraints` is given, then this argument must also be given. It is easiest to initialize this with a call to `build.constraints`. There must be rows for all of the given constraints, but additional rows may also be present.

`lin.trade` a logical vector (replicated to have length equal to the number of columns in `lin.constraints`) which determines if the constraint is to be applied to the trade (`TRUE`), or the resulting portfolio (`FALSE`). Thus the same constraint can be in the constraint matrix more than once, but they need to have different names.

`lin.abs` a logical vector (replicated to have length equal to the number of columns in `lin.constraints`). For those elements that are `TRUE`, the corresponding constraint uses the absolute value (of the weight or variance or the amount of money) for each asset (in the portfolio or trade).

WARNING: the default for this changed – it is now `TRUE`. Previous to version 1.04 the default was `FALSE`.

`lin.style` a character vector (replicated to have length equal to the number of columns in `lin.constraints`). Elements must be (an abbreviation of) one of: `"weight"`, `"value"`, `"count"`, `"varfraction"`, `"varvalue"`, `"varmbfraction"`, `"varmbvalue"`.

For "weight" constraints, the bounds are expressed in weights (position value divided by gross value of the portfolio).

For "value" constraints, the bounds are expressed in monetary units. For "count" constraints, the bounds are expressed in the number of assets; count constraints must be categorical, not numeric.

For "varfraction" and "varmbfraction", the bounds are in fractions of the variance; the latter being the variance relative to the marginal impact of the benchmark.

For "varvalue" and "varmbvalue", the bounds are in variance.

`lin.direction` a numeric vector (replicated to have length equal to the number of columns in `lin.constraints`). Allowable values are 0, 1 and -1. If 0, all assets are counted. If 1, only long assets are counted. If -1, only short assets are counted.

`lin.rfloc` a numeric vector or `NULL` (which is interpreted as 0). This gives the zero-based columns of `vtable` for the linear constraints that are of the risk fraction type. This is replicated to the number of linear constraints (number of columns in `lin.constraints`).

`alpha.constraint`

a numeric vector giving lower bounds for the alphas (expected returns), or a two-column matrix giving lower and upper bounds. If this has names, then they should coerce to integers that are one less than the column number of the desired column in the input or default `atable`. Without names, the first column (or columns) are assumed.

`var.constraint`

a numeric vector giving upper bounds for the variances, or a two-column matrix giving lower and upper bounds. If this has (row) names, then they should coerce to integers giving indices of the variance-benchmark combinations starting from zero (that is, it is one less than the column number of the combination in the input or default `vtable`). Without names, the first column (or columns) are assumed.

`bench.constraint`
: a named vector giving upper bounds for variances relative to the named benchmarks, or a two-column matrix giving lower and upper bounds. The names need to be asset names represented in the `variance` argument or the `bench.weight` argument.

`dist.center`
: either a named numeric vector, or a list with one or more components that are named numeric vectors. The names need to be asset ids like those of `prices`.

  If this is a numeric vector, then it is changed to be a length one list with that vector as its component (hence making it length one for the purposes of the descriptions of the other distance arguments). The components of this list are the target portfolios from which distances are measured.

`dist.style`
: a character vector that is replicated to have length equal to the length of `dist.center` (after possibly being coerced to a list). The elements must be (abbreviations of) strings: `"weight"`, `"value"`, `"shares"`, `"sumsqwi"`, `"customweight"`, `"customvalue"`, `"customshares"`, `"customsumsqwi"`. The "weight" style means that a vector of weights should be given in `dist.center` that sum to one. The "value" style means that the monetary values of the portfolio should be given. The "shares" style means that the number of shares (or lots or whatever) should be given. These last two mean that the distances are in terms of value rather than weight. The "sumsqwi" style means that the distance is the weighted sum of squared differences in weights where the weights are the inverse of the weight in the target portfolio (assets not in the target are ignored).

  The styles that start with "custom" all use a price vector from `dist.prices` rather than `prices` to calculate the distance.

`dist.bounds`
: a numeric vector, numeric one-column matrix, or numeric two-column matrix. A numeric vector is coerced into a one-column matrix, and a one-column matrix is coerced into a two-column matrix with the first column consisting of all zeros. The number of rows in this matrix may either be the number of components in the `dist.center` list, or that number minus the number of distances that are used in the utility (see `dist.utility`). The first column states the lower bound for the distance, and the second column states the upper bound. The numbers in each row need to correspond to the style for that distance.

`dist.trade`
: a logical vector that is replicated to be the length of `dist.center`. Values that are `TRUE` indicate that the distance should be on the trade rather than on the portfolio.

`dist.utility`
: a logical vector that is replicated to be the length of `dist.center`. Values that are `TRUE` indicate that the distance is used in the utility rather than constrained.

`dist.prices`
: either a named numeric vector or a list with components that are named numeric vectors. If a list, it can either have length one, length equal to the number of "custom" styles, or length equal to the length of `dist.center`. This gives the prices used to calculate the custom style distances.

`sum.weight`
: a numeric vector with names that correspond to integers, or a two-column matrix with row names. This constrains the sum of the largest absolute weights relative to the gross value. The name of an element (or row) gives the number of the largest weights to be summed, and the value gives the maximum (minimum and maximum in the case of a matrix)

for the sum of that number of weights. For example, to limit the sum of the 5 largest weights to be no more than 40 percent, the appropriate `sum.weight` argument would be: `c("5"=.4)` Note the quotes around the 5.

`limit.cost` either `NULL` or a length 2 numeric vector giving the lower and upper bounds on the cost. This is useful for random portfolios, but is unlikely to be of interest for optimization.

`close.number` if given, a numeric vector of length 1 or 2. This is the lower and upper bounds for the number of positions to close in the existing portfolio. If of length 1, then exactly that number of positions are to be closed.

`utility` character string; the possibilities are (the start of): `"information ratio"`, `"mean-variance"`, `"exocost information ratio"`, `"minimum variance"`, `"maximum return"`, `"mean-volatility"`, `"distance"`. The information ratio is the expected return of the portfolio divided by the square root of the variance of the portfolio. The default is the information ratio if both `variance` and `expected.return` are given and `dist.utility` is not `TRUE`. Otherwise, it infers what the default must be. This argument is ignored if `utable` is given.

`risk.aversion` number giving the risk aversion for the mean-variance or mean-volatility utility. The utility is the risk aversion times the variance (or volatility) of the portfolio minus the expected return of the portfolio. Many other formulations use one-half the variance, so the risk aversion is different by a factor of two. If you want infinite risk aversion, you should set the objective to `"minimum variance"`. This argument is ignored except in the two cases where risk aversion appears in the utility.

`benchmark` vector of one or more character strings naming the benchmark(s). These must appear in the variance but need not appear as names in `prices` unless `bench.trade` is `TRUE`.

`bench.trade` logical value. If `TRUE`, then assets identified as benchmarks (in argument `benchmark` and `bench.constraint`) can be traded (and hence need to have a valid price). This is effectively set to `FALSE` if `universe.trade` is given.

`bench.weights` a list where the name of each component is the name of a benchmark. Each component should be a named numeric vector that sums to 1 (or 100). All of the names on the weights need to be in `variance`. Benchmarks in this argument are added to `variance` if they are not included in it. Otherwise the variance is checked to see if the values in the variance are implied by the weights; if not, then the variance values are retained but a warning is issued unless `do.warn` item `variance.benchmark` is `FALSE`. The same is true for `expected.returns` except the `do.warn` item is `alpha.benchmark`.

`long.buy.cost` a vector or matrix of the cost to buy assets if they are long. The meaning depends on whether or not `cost.par` has length.

If `cost.par` is `NULL`: If a vector (or single column matrix), the costs are linear; that is, the cost in the objective for an asset is the number bought times the corresponding element in `buy.cost`.

If this is a matrix with more than one column and `cost.par` has zero length, then it represents a polynomial of order one less than the number of columns. The first column is the intercept, the second column corresponds to number of units traded (to the first power), the third column corresponds to number of units squared, etc.

If `cost.par` has positive length: Then this must have as many columns as the length of `cost.par`. Each column is the coeficient of units to the power given by the corresponding element of `cost.par`.

If this has names (row names in the case of a matrix), then it may contain information for assets that are not in the current problem. However, it must have information on all of the tradable assets that are in the current problem.

`long.sell.cost`

a vector or matrix of the cost to sell long assets. See `long.buy.cost` for a full description. The default value of this is the value of `long.buy.cost`.

`short.buy.cost`

a vector or matrix of the cost to buy short assets. See `long.buy.cost` for a full description. The default value of this is the value of `long.buy.cost`.

`short.sell.cost`

a vector or matrix of the cost to sell short assets. See `long.buy.cost` for a full description. The default value of this is the value of `long.buy.cost`.

`cost.par` `NULL` or a vector or matrix of exponents to the number of units for trading costs. If this is given, then all four of the arguments giving cost values must have the same number of columns as the length of `cost.par` (if it is a vector) or the number of columns of `cost.par` (if it is a matrix).

Not giving this argument is equivalent to giving a sequence of integers (that starts at zero if there is more than one column in the costs).

If this is a matrix, then it should have row names that identify the assets. This form allows each asset to have its own individual exponents in the cost function.

`ucost` a number that scales the trading costs in the utility. For example, when maximizing returns, what is really maximized is the portfolio return minus `ucost` times the transaction costs.

`scale.cost` a character string indicating what scaling should be done on the costs. This must partially match one of: `"gross"` (the most likely choice and the default), `"trade"`, `"none"`.

`start.sol` there are three possibilities (apart from `NULL`): (1) an object resulting from `trade.optimizer`; (2) a named numeric vector (such as the `trade` component of the output of a previous optimization); (3) a list of such vectors. The starting solutions, which need not have length equal to `ntrade`, are used as suggestions for the optimizer.

If a single starting solution is given and `funeval.max` is set to 0 or 1, then the result will be for the starting solution as the trade – no actual optimization is performed.

`allowance` number less than one (but generally close to one) which provides the allowable range of `gross.value`, `net.value`, `long.value` and `short.value` if only a single number is given for the argument.

`do.warn` logical vector. The most likely use is to give a vector of all `FALSE` values with names. The names need to partially match: `"cost.intercept.nonzero"`, `"converged"`, `"value.range"`, `"extraneous.assets"`, `"no.asset.names"`, `"benchmark.long.short"`, `"variance.list"`, `"turnover.max"`, `"max.weight.restrictive`
`"neg.dest.wt"`, `"penalty.size"`, `"noninteger.forced"`, `"ignore.max.weight"`, `"bounds.missing"`, `"positions.names"`, `"start.noexist"`, `"random.start"`, `"novariance.optim"`, `"nonzero.penalty"`, `"neg.risk.aversion"`, `"zero.iterations"`, `"infinite.bounds"`, `"notrade"`, `"randport.failure"`, `"utility.switch"`,

"thresh.notrade", "dist.prices", "dist.style", "dist.zero", zero.variance, alpha.benchmark, variance.benchmark, var.eps, back.compat, index.zero, riskfrac.part, exit.obj, superfluous.constraint. Warning messages will be suppressed when their type is set to FALSE. The default values are all TRUE except for "converged". Some warnings are never suppressed.

penalty.constraint
: a numeric vector giving the multiplier for each penalty. This is replicated to have length equal to the number of columns in constraint.matrix plus the number of variance constraints, etc.

quantile
: a number between zero and one (inclusive) giving the quantile from the utilities when there is more than one utility destination. In general, it only makes sense to use numbers that are one-half or greater. When quantile is one, then this is a min-max solution; that is, for each portfolio we look at the worst (minimum) utility over the specified combinations of expected returns, variances and benchmarks, then we select the portfolio that maximizes the worst utility. With quantile set to one-half, we are getting the median utility over the combinations of expected returns, variances and benchmarks.

dest.wt
: numeric vector giving strictly positive weights to the destinations of the combinations of expected returns and variances. When this is given, then the quantile argument refers to a weighted quantile.

utable
: a numeric matrix which controls the combinations of expected returns, variances and benchmarks, and their placement into the destinations. Providing this is only necessary if there are multiple expected returns, variances or benchmarks, and the default treatment of them is not what is desired. The matrix has 6 rows: "alpha.spot" is the (zero-based) index of the columns of atable; "variance.spot" is the zero-based index of columns of vtable; "destination" is the zero-based index of the destination; "opt.objective" is 0 for a mean-variance utility, 1 for an information ratio utility, 2 for exocost information ratio, 3 for minimum variance, 4 for maximum return, 5 for mean-volatility, 6 for distance; "risk.aversion" is the risk aversion; and "wt.in.destination" is the weight of this combination inside the destination for this combination.

atable
: numeric matrix with two rows describing the combinations of expected returns and benchmarks. The first row is the zero-based index of the expected returns (that is, one less than the column index of expected.return). The second row is the index within the assets of the benchmark (this starts numbering from zero); a negative number indicates no benchmark. However if given, then the second row is ignored and is created by the benchmarks attribute that is required. This attribute needs to be a character vector as long as the number of columns in atable; the elements are either asset names or the empty string (meaning no benchmark).

vtable
: numeric matrix with three rows describing the combinations of variances and benchmarks. The first row is the zero-based index of the variance matrix (that is, one less than the index of the third dimension of variance). The second row is the index within the assets of the benchmark (this starts numbering from zero); a negative number indicates no benchmark. However if given, then the second row is ignored and is created by the benchmarks attribute that is required. This attribute needs to be a character vector as long as the number of columns in vtable; the elements

are either asset names or the empty string (meaning no benchmark). The third row is one or zero depending on if the variance-benchmark combination is used in the utility or not.

`dumpfile` a character string giving the name of a file to be produced for the purposes of debugging in the event of certain errors being triggered due to a bug. This is hopefully a superfluous argument. No file is created by default.

`...` individual arguments to `trade.optimizer.control` (or whatever other function is given as `control`) may be given if `control` is not specifically given as a list.

`seed` the seed for the random number generator that is internal to the optimizer. The whole name ("seed") must be used – no abbreviations allowed. If this is given, it will most likely be `NULL` or a single integer to create a random seed via `seed.BurSt`. Specifying this produces a random result as opposed to the default behavior of using a particular seed (`.standard.seed.BurSt`).

`control` a list like the output of `trade.optimizer.control` that sets numerous controls for the optimizer. Alternatively, it can be a function that produces such a list. The whole name ("control") must be used – no abbreviations allowed.

`identity` an arbitrary object, most likely a string or an integer. This is merely added unchanged to the output. It is useful when an optimization is repeatedly done in parallel. The whole name ("identity") must be used – no abbreviations allowed.

Value

an object of class `portfolBurSt` which is a list with the following components:

`new.portfolio`
named numeric vector giving the optimal portfolio. The numbers are the number of units (shares, lots) of the assets.

`trade` named numeric vector giving the trade needed to achieve the optimal portfolio. The numbers are the number of units (shares, lots) of the assets.

`results` numeric vector containing various results from the optimization. It contains the objective achieved, the negative utility, the trading cost, and the penalty (for unmet constraints) component of the objective. The objective is the sum of the negative utility and the penalty. The cost is included (possibly non-linearly) in the negative utility.

`converged` logical value(s): if a sufficiently large number of consecutive iterations failed to improve the solution, then convergence is declared to have occurred. If `stringency` is positive, then two values are returned: the first element (called `"total"`) is `TRUE` if the stringency requirement of finding enough identical solutions is satisfied; the second element is the convergence status of the final run performed.

`objective.utility`
character string stating the utility that is the objective of the optimization.

`universe.size`
a vector giving the number of assets in the universe, the number of assets allowed to trade, the number that the `universe.trade` argument allows to trade, and the number that the `positions` argument allows to trade.

This component is suppressed when the object is printed, but it is a part of the output of the `summary` method for optimized portfolios.

`utable` the utility table that was input or created for the problem. This component is suppressed when the object is printed.

`atable` the input or created `atable`. This component is suppressed when the object is printed.

`vtable` the input or created `vtable`. This component is suppressed when the object is printed.

`alpha.values` vector with length equal to the number of expected return-benchmark combinations giving the (ex-ante) expected return(s) of the optimal portfolio.

The values assume that the net asset value is the gross value. Scaling is called for if that assumption is not correct.

`var.values` vector with length equal to the number of variance-benchmark combinations giving the (ex-ante) variance(s) of the optimal portfolio.

The values assume that the net asset value is the gross value. Scaling is called for if that assumption is not correct.

`utility.values`

vector of the negative utilities (including costs and penalties) of the optimal portfolio. If there are multiple values, these are sorted – they are not in the same order as the columns of `utable` or destinations.

`constraint.violations`

numeric vector of the violations of each constraint type. Zero means the constraint is satisfied. This component is suppressed when the object is printed.

`penalty.constraint`

numeric vector of the penalties for each constraint type. This component is suppressed when the object is printed.

`value.limits` two-column matrix giving the lower and upper bounds of the gross value, net value, long value and short value. This component is suppressed when the object is printed, but is shown in the summary.

`prices` the vector of prices of relevant assets used in the optimization. That is, prices relevant to the solution found. This component is suppressed when the object is printed.

`optim.mumbo.jumbo`

numeric vector of results of the optimization. The first four values pertain to the final run (which is the only run unless `stringency` is positive). These give: number of iterations performed, the number of iterations to have improved the solution, the final number of consecutive iteration failures, and the maximum number of consecutive iteration failures.

The next group refers to the entire optimization. These give: the number of portfolios evaluated, the total number of requested evaluations, the number of evaluations requested that were amusingly silly, and a flag denoting the state of the result.

Finally there are values that are relevant when `stringency` is positive: the number of runs, the total number of iterations, the total number of successes and the number of assets that were in the trades of the runs in the initial phase.

This component is suppressed when the object is printed, but is shown in the summary.

`risk.fraction`
: if argument `risk.fraction` is given or there are variance fraction style linear constraints, then this component will be a matrix containing the fractions of variance for the assets. This component is suppressed when the object is printed.

`existing`
: the existing portfolio before the optimization.

`violated`
: character vector giving the types (if any) of constraints that are broken by the solution.

`seed`
: the random seed for the internal random number generator of the optimizer. This component is suppressed when the object is printed.

`version`
: character vector giving the version numbers for the C code and the S language code. This component is suppressed when the object is printed.

`sizes`
: numeric vector describing aspects of the problem – generally uninteresting. This component is suppressed when the object is printed.

`dist.table`
: a data frame giving information on each distance. The columns are: `style` (character); `trade` logical stating if the distance is for the trade or portfolio; `utility` logical stating if the distance is a utility or a constraint; and then the three numeric columns `lower.bnd`, `upper.bnd`, `value`.

`dist.center`
: a list giving the target portfolio(s). This component is suppressed when the object is printed.

`dist.prices`
: either `NULL` or a list giving the custom distance prices. This component is suppressed when the object is printed.

`benchmarks`
: vector giving any benchmarks used. The names of the vector are the names of the benchmarks. The values in the vector are the zero-based indices of the benchmarks as used in `atable` and `vtable`. This component is not present if there are no benchmarks. This component is suppressed when the object is printed.

`forced.explicit`
: vector giving the trades explicitly forced using the `forced.trade` argument. This component is not present if no trades were explicitly forced.

`positions.forced`
: vector giving the trades that were forced via the `positions` argument. This component is not present if there were no such trades.

`all.forced`
: vector giving the trades that were forced – including trades implicitly forced in order to try to satisfy maximum weight constraints, and trades forced via the `positions` argument. This component is not present if no trades were forced.

`positions`
: the input `positions` if given. This component is suppressed when the object is printed.

`tol.positions`
: the input `tol.positions` if `positions` is given. This component is suppressed when the object is printed.

`lin.table`
: a data frame showing information on the set of linear constraints, one row for each (top-level) constraint. The columns are:

  `style`: states the style of the constraint (`"varfraction"`, `"count"`, `"weight"`, etc.).

  `trade`: logical stating whether the constraint is for the trade or the portfolio.

`absolute`: logical stating whether the constraint is for the gross or net.

`levels`: the number of levels for the constraint; this is zero for numeric constraints.

`direction`: 0 for all assets, 1 for long-only constraints, -1 for short-only constraints.

`rfloc`: the zero-based column of `vtable` to which a variance fraction style constraint refers; this is -1 for other constraint styles.

`riskfrac.col`: the column of the `risk.fraction` matrix (in the output) that a variance fraction style constraint uses; this is 0 for other constraint styles.

This component is not present unless `lin.constraints` was given. This component is suppressed when the object is printed, but is shown in the summary.

`con.realized` a list with 0 or 1 components. The possible component is:

`linear`: matrix giving bounds, values and violations of the linear constraints.

This component is suppressed when the object is printed, but is shown in the summary.

`iterhistory` a vector giving the objective value at each iteration of the optimization process. The length of this is two more than the number of iterations done: the first value gives the objective after the pre-iteration phase, and the last value gives the objective after the post-iteration phase. This is only present if the `save.iterhistory` control parameter is `TRUE`. In general, this is only of interest when investigating the quality of optimizations.

`identity` the input value of `identity`. This component is suppressed when the object is printed.

`checkinput` a list containing components: `prices`, `variance`, `expected.return` and `existing` with a few values each of those inputs. This is used in particular with random portfolios and updating to try to make sure that no sleight of hand is taking place with data. This component is suppressed when the object is printed.

`timestamp` two character strings giving the time and date of creation (the time the command started and the time it ended).

`call` an image of the call that created the object.

Side Effects

The S language random seed is changed or created if the `seed` argument is such that it generates a new seed. That is not the default.

Details

SPECIFYING THE MONETARY VALUE. While `prices` is always required, there is no other single argument that is always required. However, there is the requirement that the value of the portfolio be specified somehow. For any problem, specifying `turnover` is enough. For long-only problems, any one of `gross.value`, `long.value` or `net.value` being specified works. For long-short portfolios, then either `gross.value` and `net.value` need to be specified, or `long.value` and `short.value` (assuming that `turnover` has not been given). If any of the gross, net, long or short values are given, then at least one pair – either gross and net, or long and short – need to be given.

BOUNDING THE TRADE. The `lower.trade` and `upper.trade` arguments bound the number of units that may be traded for each asset. The optimizer insists that the range of allowable units includes zero. But you can use the `forced.trade` argument to specify trades that must be performed. You can also give the `positions` argument which is in monetary value rather than shares (or lots). `positions` can give limits on the value in the portfolio and the value to trade including forced trades; it can also specify assets not to trade (by placing `NA` in at least one of the first four columns of that row).

THE PENALTY. The objective not only includes the negative utility (including the cost of the trade) but also a penalty for breaking constraints. If the penalty is non-zero, then the optimization should generally be redone, or the constraints changed – the portfolio may not be very optimal when penalties are imposed. There are times when trivial penalties are imposed by constraints that are at the boundary – the optimality will not be affected in such cases.

SPEED AND OPTIMIZATION QUALITY. There is a trade-off between the time the optimizer takes and the quality of the optimization that you are likely to get. The control arguments `iterations.max` and `fail.iter` can be increased to improve the solution, or decreased to speed the optimization. The quality can also be improved by increasing `stringency` from zero. The User's Manual has more on this.

Bugs

If the existing portfolio breaks maximum weight constraints (from the `max.weight` argument), there are situations in which the optimal portfolio will not satisfy those constraints – see the User's Manual for more information.

For long-only portfolios if more than one of `gross.value`, `net.value`, `long.value` is given, then the range of all these values is used as the allowable interval. More logical (but much more bother) would be the intersection of the intervals. Best is to just give one of these.

The `trace` argument has no effect under Windows in some cases – tracing information on this platform is given in batch mode.

Revision

This help was last revised 2012 June 10.

See Also

`trade.optimizer.control`, `build.constraints`, `random.portfolio`, `summary.portfolBurSt`, `deport.portfolBurSt`, `seed.BurSt`, `constraints.realized`, `Cfrag.list`.

Examples

```
# build long-only portfolio worth 1 million containing no more than 60 assets
op1 <- trade.optimizer(eq.prices, varian, long.only=TRUE,
        gross.value=1e6, ntrade=60)
summary(op1)
deport(op1) # csv file of the trade

# revise a long-short portfolio, keep the number of assets no more than 50
op2 <- trade.optimizer(eq.prices, varian, gross.value=1e6,
        net.value=c(-1e3, 5e3), exist=cur.port, port.size=50)
```

`trade.optimizer.pre` *Select an Optimal Trade for a Portfolio*

Description

Function that is a utility for `trade.optimizer` and `random.portfolio`. This is not meant for direct use.

Usage

```
trade.optimizer.pre(...)
```

Arguments

`...`	See the description of arguments in `trade.optimizer`.

Value

a list containing objects to be used in computing the problem.

Side Effects

none.

Revision

This help was last revised 2012 April 17.

See Also

`trade.optimizer`, `trade.optimizer.control`, `random.portfolio`.

`trade.optimizer.control`
Optimizer Controls for Trade Selection

Description

Sets parameters that control the optimization algorithm in `trade.optimizer` (and hence in `random.portfolio.utility`).

Usage

```
trade.optimizer.control(iterations.max = 20, fail.iter = 0,
        funeval.max = .Machine$integer.max, trace = TRUE,
        exit.obj = -big, runs.init = 3, runs.final = 2,
        runs.min = 1, stringency = 0, nonconverge.mult = 2,
        feasible = 0, miniter = 0, force.risk.aver = FALSE,
        enforce.max.weight = TRUE, save.iterhistory = FALSE,
        safe.mode = TRUE, ...)
```

Arguments

`iterations.max`
: integer giving the maximum number of iterations to perform in a run.

`fail.iter`
: integer giving the maximum number of consecutive iterations that fail to improve the solution without the algorithm stopping and declaring that convergence has been achieved. For example, if `fail.iter` is 5, then the algorithm continues with 5 consecutive failures but stops upon the 6th consecutive failure.

`funeval.max`
: integer giving the maximum number of function (portfolio) evaluations to perform. The most common use of this is to set it to one or zero when you have a specific trade that you would like to have done.

`trace`
: logical value. If `TRUE`, then information on the progress of the optimization is printed. This is ignored under Windows – tracing information is not available on this platform except in BATCH mode.

`exit.obj`
: number such that the optimization terminates if the best solution has an objective at least as good (i.e., an objective smaller than or equal to `exit.obj`). (Useful for comparing algorithms, but otherwise probably uninteresting.)

`runs.init`
: if `stringency` is greater than zero, then the maximum number of runs to do on the original problem.

`runs.final`
: if `stringency` is greater than zero, then the maximum number of runs to do with the trading universe restricted to those assets that traded in at least one of the initial runs.

`runs.min`
: if `stringency` is greater than zero, then the minimum number of runs that are to be performed.

`stringency`
: an integer giving one less than the number of runs that need to match as the best solution before the optimization is exited. If `stringency` is 0, then only one run is done. If `stringency` is 1, then the best solution needs to occur twice before the optimization stops. If the best solution does not occur enough times before the maximum number of runs is exhausted, then a final non-convergence run is performed that starts from the best solution found.

`nonconverge.mult`
: if `stringency` is greater than zero and the stringency requirement is not met, then a final run is performed starting with the best solution found. `nonconverge.mult` gives the ratio of iterations allowed for this final run relative to the iterations allowed in the others. So if `iterations.max` is 5 and `nonconverge.mult` is 3, then the non-convergence run is allowed up to 15 iterations.

`feasible`
: if all constraints are not met at the end of the iteration equal to the value of `feasible`, then the optimization is aborted. But if `feasible` is 0, then the optimization does not stop on account of broken constraints.

`miniter`
: the minimum number of iterations to be performed in a run even if `fail.iter` says to stop already.

`force.risk.aver`
: logical value which is only used when the `utable` argument is explicitly given and optimization with risk aversion is performed. If `TRUE`, then the `risk.aversion` argument overrides the risk aversion values in the utility table.

`enforce.max.weight`
: logical value; if `TRUE`, then forced trades are automatically created if any positions in the existing portfolio break their maximum weight constraint. This doesn't absolutely guarantee that the constraints will be met, but they generally will be unless the gross value is given a lot of latitude.

`save.iterhistory`
: logical value; if `TRUE`, then a vector of the objective value at each iteration of the optimization will be returned.

`safe.mode`
: logical value; if `FALSE` (not recommended), then some errors may be bypassed. The reason for this option is to allow possible workarounds for bugs if they appear.

`...`
: there are additional arguments that make this function compatible with `random.portfolio.control` (because the two tasks use the same code internally).

Value

a list with the following components:

`icontrol`
: vector of the integer-valued control parameters.

`dcontrol`
: vector of the double precision control parameters.

`aux`
: vector of the auxiliary control parameters.

Revision

This help was last revised 2012 April 17.

See Also

`trade.optimizer`, `random.portfolio.utility`, `random.portfolio.control`.

Examples

```
my.to.control <- trade.optimizer.control(iterations=100, fail=5,
        stringency=1, runs.init=5, runs.final=3)
trade.optimizer(eq.prices, varian, control=my.to.control)
```

`update.randportBurSt` *Re-execute a random portfolio object*

Description

Re-executes a call to `random.portfolio`, possibly with some arguments changed.

Usage

```
update.randportBurSt(object, ..., evaluate = TRUE,
        checkinput = TRUE, envir = parent.frame())
```

Arguments

object	object produced by random.portfolio (or random.portfolio.utility).
...	arguments to random.portfolio that should be added or changed from those in the initial call.
evaluate	logical value. If TRUE (the default), then the command is executed. If FALSE, then the (revised) call is returned. This argument needs to be given by its full name since it comes after the three-dots in the argument list.
checkinput	logical value. If TRUE (the default), then a check is made to see if the prices, variance, expected return and the existing portfolio are the same in the new call as the original call. Except that any of these are excluded from the test if they are arguments in the call to update. This argument needs to be given by its full name since it comes after the three-dots in the argument list.
envir	the environment in which the evaluation is to take place.

Value

If evaluate is TRUE, then the possibly revised call is executed, producing a random portfolio object (of class "randportBurSt").

Details

Arguments that are changed need to use the exact same abbreviation (if any) as the original call. For instance, if in the original call net.value is abbreviated to net.val, then you need to use net.val if you want to change the net value allowed.

The default method of update works for the results of trade.optimizer so it doesn't need a special method. However, this means that there is no check for the key inputs changing. If you want such a check, then you need to do it yourself as can be seen in the help file for pprobe.checkinput.

Revision

This help was last revised 2012 April 17.

See Also

random.portfolio, trade.optimizer, pprobe.checkinput.

Examples

```
randport1 <- random.portfolio(100, prices, variance=varian,
        long.only=TRUE, bench.constr=c(spx=.04^2/252),
        lin.constraints=cntrysect.conmat,
        lin.bounds=cntrysect.bounds, gross.value=1e6)

randport2 <- update(randport1) # same again
randport3 <- update(randport1, gross.value=1.5e6)

opt1 <- trade.optimizer(prices, variance=varian,
        long.only=TRUE, bench.constr=c(spx=.04^2/252),
        lin.constraints=cntrysect.conmat,
        lin.bounds=cntrysect.bounds, gross.value=1e6)
```

```
opt2 <- update(opt1, gross.value=1.5e6)

call1 <- update(opt1, number.rand=100, evaluate=FALSE)
call1[[1]] <- as.name("random.portfolio")
randport4 <- eval(call1) # random portfolios like opt1
```

`valuation.portfolBurSt`

Find the Monetary Values of a Portfolio or Trade

Description

Returns various concepts of value.

Usage

```
## S3 method for class 'portfolBurSt':
valuation(x, prices = x$prices, trade = FALSE,
        weight = is.logical(collapse) && !collapse,
        collapse = is.array(prices), type = "gross", cash = NULL,
        all.assets = FALSE, returns = NULL, drop.factor = FALSE)
## Default S3 method:
valuation(x, prices,
        weight = is.logical(collapse) && !collapse,
        collapse = is.array(prices), type = "gross", cash = NULL,
        all.assets = FALSE, returns = NULL, drop.factor = FALSE)
```

Arguments

`x`	required. An object of class `"portfolBurSt"` created by `trade.optimizer` (for the `"portfolBurSt"` method). Or a named vector (for the default method).
`prices`	a named vector of asset prices that includes at least the assets in the object being evaluated; or a matrix with column names for the assets (rows are for times); or a three-dimensional array with rows corresponding to times, columns to assets, and slices to different scenarios. This is a required argument for the default method.
`trade`	logical value. If `TRUE`, then the results are for the trade. If `FALSE`, then the results are for the optimal portfolio.
`weight`	logical value; if `TRUE` and `collapse` is `FALSE`, then the result includes a component named `weight` which contains the weights of the assets (that is, position values divided by the gross value). If `weight` is `TRUE` and `collapse` represents categories, then the results are weights within the categories (or category combinations).
`collapse`	logical value, or an object representing categories. If `TRUE`, then the result will be a number (or a vector if `prices` was a three-dimensional array) containing results for the portfolio rather than individual assets. Categories can be represented by a named character vector or factor, or a list of these (for combinations of categories). The list can be in the form of a data frame where the row names are the asset identifiers.

`type` this is only used when `collapse` is not `FALSE`.

A character string that (partially) matches one of: `"gross"`, `"net"`, `"long"`, `"short"`, `"nav"`, `"cash"`. When this has value `"nav"`, then the result is the net value plus cash. When this has value `"short"`, the result is non-negative numbers. The `"cash"` type lets you see the value of cash that is computed.

`cash` this is only used when `type="nav"` and `collapse` is `TRUE`.

A single number, a numeric vector with as many elements as there are rows in `prices`, or `NULL`. If this is `NULL`, then the amount of cash is such that the initial value of the portfolio is equal to the gross value minus the net value at the first time point.

`all.assets` a logical value. if `TRUE`, then the `weight` and `individual` components will have the same assets (in the same order) as `prices`.

`returns` either `NULL` or a single character string that is (an abbreviation of) `"log"`, `"simple"`, `"geometric"`, or `"arithmetic"`. This fails if `prices` is not a matrix or array. This is two types of returns with two names each.

`drop.factor` a logical value. if `TRUE`, then (combinations of) categories in `collapse` that do not occur are not represented in the result.

Value

if `collapse` is `TRUE`, then an object with dimension one less than `prices`. The order of the dimensions is: time (if any), and scenarios (if any).

If `collapse` represents categories, then an object with the same number of dimensions as `prices` (and currently `prices` is not allowed to be three-dimensional in this case). The order of dimensions is: time (if any) and categories.

The above include the attributes `timestamp` and `call` like the descriptions given below.

If `collapse` is `FALSE`, a list with the following components:

`individual` the amount of money in individual assets.

`total` length 4 vector containing the gross value, the net value, the long value and the short value.

`weight` only present if the `weight` argument was `TRUE`. A numeric vector containing the weights of the assets.

`timestamp` character string giving the time and date of the evaluation.

`call` the call that created the object.

Details

These are methods for the generic function `valuation`.

Revision

This help was last revised 2012 February 24.

See Also

`trade.optimizer`, `summary.portfolBurSt`, `valuation.randportBurSt`.

Examples

```
op1 <- trade.optimizer(eq.prices, varian, long.only=TRUE,
        gross.value=1e6)
valuation(op1)

valuation(op1, new.prices) # valuation with new prices

valuation(op1, rbind(eq.prices, new.prices)) # two gross values

valuation(op1, trade=TRUE) # valuation for the trade
valuation(op1$trade, eq.prices) # same thing

# get weight vector corresponding to full universe of assets
valuation(op1, pricevec, all.assets=TRUE)$weight

# log returns over some time period
valuation(op1, price.matrix, returns="log")
```

---

valuation.randportBurSt

Monetary Value of Random Portfolios

Description

Returns one of various concepts of value for the portfolios.

Usage

```
## S3 method for class 'randportBurSt':
valuation(x, prices, weight = FALSE,
        collapse = is.array(prices), type = "gross", cash = NULL,
        all.assets = FALSE, returns = NULL)
```

Arguments

`x`	required. An object of class `"randportBurSt"` (generally the result of a call to `random.portfolio`).
`prices`	required. A named vector of asset prices that includes at least the assets in the object being evaluated; or a matrix with column names for the assets (rows are for times); or a three-dimensional array with rows corresponding to times, columns to assets, and slices to different scenarios.
`weight`	logical value; if `TRUE`, then the result will contain weights rather than monetary value.
`collapse`	logical value, or an object representing categories. If `TRUE`, then the result will be a vector or matrix containing results for the portfolios rather than individual assets. Categories can be represented by a named character vector or factor, or a list of these (for combinations of categories). The list can be in the form of a data frame where the row names are the asset identifiers.

type	this is only used when collapse is not FALSE. A character string that (partially) matches one of: "gross", "net", "long", "short", "nav", "cash". When this has value "nav", then the result is the net value plus cash. When this has value "short", the result is non-negative numbers. The "cash" type lets you see the value of cash that is computed.
cash	this is only used when type="nav" and collapse is not FALSE. A single number, a numeric vector with as many elements as there are rows in prices, or NULL. If this is NULL, then the amount of cash is such that the initial value is equal to the gross value minus the net value at the first time point for each portfolio.
all.assets	a logical value. if TRUE, then the weight and individual components will have the same assets (in the same order) as prices.
returns	either NULL or a single character string that is (an abbreviation of) "log", "simple", "geometric", or "arithmetic". This fails if prices is not a matrix or array. This is two types of returns with two names each.

Value

a list, vector, matrix or three-dimensional array containing valuation information, and with some attributes.

If collapse is FALSE, then the result is a list of the same length as the input x where each component holds the asset valuations or weights.

If collapse is TRUE, then the result is an object with the same number of dimensions as prices, but the dimension in prices that corresponds to assets corresponds to random portfolios in the result.

if collapse represents categories, then the result has one more dimension than prices (and currently prices is not allowed to be three-dimensional in this case). The order of dimensions is: time (if any), categories, random portfolios.

The (extra) attributes are:

timestamp	character string giving the date and time of the evaluation.
call	the call that created the object.

Details

This is a method for the generic function valuation for class randportBurSt.

If the portfolios are long-short and the valuations are going to be used to calculate returns, then you should set type="nav" and set cash properly for the situation that you have.

If out.trade is TRUE in the call for x, then the results are for the trades and not for the portfolios.

Revision

This help was last revised 2012 February 24.

See Also

random.portfolio, deport.randportBurSt, valuation.portfolBurSt.

Examples

```
randport1 <- random.portfolio(100, prices, varian, long.only=TRUE,
        bench.constr=c(spx=.04^2/252), lin.constraints=cntrysect.conmat,
        lin.bounds=cntrysect.bounds, gross.value=1e6)

randval1 <- valuation(randport1, prices)
randval1n <- valuation(randport1, new.prices)

randret <- valuation(randport1, price.matrix, returns="simple")

# weights by country-sector combination
randwgt.cntrysect <- valuation(randport1, price.matrix,
        collapse=list(country, sector), weight=TRUE)
```

Index