Package: caTools (via r-universe)

September 5, 2024

Type Package

Title Tools: Moving Window Statistics, GIF, Base64, ROC AUC, etc

Version 1.18.3

Date 2024-09-04

Maintainer Michael Dietze <michael.dietze@uni-goettingen.de>

Depends R (>= 3.6.0)

Imports bitops

Suggests MASS, rpart

Description Contains several basic utility functions including: moving (rolling, running) window statistic functions, read/write for GIF and ENVI binary files, fast calculation of AUC, LogitBoost classifier, base64 encoder/decoder, round-off-error-free sum and cumsum, etc.

License GPL-3

NeedsCompilation yes

Author Jarek Tuszynski [aut], Michael Dietze [cre]

Date/Publication 2024-09-04 17:00:02 UTC

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

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

RemoteRef HEAD

RemoteSha 909332fbe26309720f68448906b57fe1e1ddf7ad

Contents

`caTools-package` *Tools: moving window statistics, GIF, Base64, ROC AUC, etc.*

Description

Contains several basic utility functions including: moving (rolling, running) window statistic functions, read/write for GIF and ENVI binary files, fast calculation of AUC, LogitBoost classifier, base64 encoder/decoder, round-off error free sum and cumsum, etc.

Author(s)

Jarek Tuszynski <jaroslaw.w.tuszynski@saic.com>

Examples

```
# GIF image read & write
write.gif( volcano, "volcano.gif", col=terrain.colors, flip=TRUE,
         scale="always", comment="Maunga Whau Volcano")
y = read.gif("volcano.gif", verbose=TRUE, flip=TRUE)
image(y$image, col=y$col, main=y$comment, asp=1)
file.remove("volcano.gif")

# test runmin, runmax and runmed
k=25; n=200;
x = rnorm(n,sd=30) + abs(seq(n)-n/4)
col = c("black", "red", "green", "brown", "blue", "magenta", "cyan")
plot(x, col=col[1], main = "Moving Window Analysis Functions (window size=25)")
lines(runmin (x,k), col=col[2])
lines(runmed (x,k), col=col[3])
lines(runmean(x,k), col=col[4])
lines(runmax (x,k), col=col[5])
legend(0,.9*n, c("data", "runmin", "runmed", "runmean", "runmax"), col=col, lty=1 )

# sum vs. sumexact
x = c(1, 1e20, 1e40, -1e40, -1e20, -1)
a = sum(x);         print(a)
b = sumexact(x);    print(b)
```

`base64encode & base64decode` *Convert R vectors to/from the Base64 format*

Description

Convert R vectors of any type to and from the Base64 format for encrypting any binary data as string using alphanumeric subset of ASCII character set.

Usage

```
base64encode(x, size=NA, endian=.Platform$endian)
base64decode(z, what, size=NA, signed = TRUE, endian=.Platform$endian)
```

Arguments

`x` vector or any structure that can be converted to a vector by `as.vector` function. Strings are also allowed.

`z` String with Base64 code, using [A-Z,a-z,0-9,+,/,=] subset of characters

`what` Either an object whose mode will give the mode of the vector to be created, or a character vector of length one describing the mode: one of ’"numeric", "double", "integer", "int", "logical", "complex", "character", "raw". Same as variable `what` in `readBin` functions.

`size` integer. The number of bytes per element in the byte stream stored in `r`. The default, ’`NA`’, uses the natural size. Same as variable `size` in `readBin` functions.

`signed` logical. Only used for integers of sizes 1 and 2, when it determines if the quantity stored as raw should be regarded as a signed or unsigned integer. Same as variable `signed` in `readBin` functions.

`endian` If provided, can be used to swap endian-ness. Using ’"swap"’ will force swapping of byte order. Use ’"big"’ (big-endian, aka IEEE, aka "network") or ’"little"’ (little-endian, format used on PC/Intel machines) to indicate type of data encoded in "raw" format. Same as variable `endian` in `readBin` functions.

Details

The Base64 encoding is designed to encode arbitrary binary information for transmission by electronic mail. It is defined by MIME (Multipurpose Internet Mail Extensions) specification RFC 1341, RFC 1421, RFC 2045 and others. Triplets of 8-bit octets are encoded as groups of four characters, each representing 6 bits of the source 24 bits. Only a 65-character subset ([A-Z,a-z,0-9,+,/,=]) present in all variants of ASCII and EBCDIC is used, enabling 6 bits to be represented per printable character.

Default `size`s for different types of `what`: `logical` - 4, `integer` - 4, `double` - 8 , `complex` - 16, `character` - 2, `raw` - 1.

Value

Function base64encode returns a string with Base64 code. Function base64decode returns vector of appropriate mode and length (see x above).

Author(s)

Jarek Tuszynski (SAIC) <jaroslaw.w.tuszynski@saic.com>

References

- Base64 description in *Connected: An Internet Encyclopedia* https://www.freesoft.org/CIE/RFC/1521/7.htm
- MIME RFC 1341 http://www.faqs.org/rfcs/rfc1341.html
- MIME RFC 1421 http://www.faqs.org/rfcs/rfc1421.html
- MIME RFC 2045 http://www.faqs.org/rfcs/rfc2045.html
- Portions of the code are based on Matlab code by Peter Acklam

See Also

xmlValue from **XML** package reads XML code which sometimes is encoded in Base64 format.

readBin, writeBin

Examples

```
x = (10*runif(10)>5) # logical
for (i in c(NA, 1, 2, 4)) {
  y = base64encode(x, size=i)
  z = base64decode(y, typeof(x), size=i)
  stopifnot(x==z)
}
print("Checked base64 for encode/decode logical type")

x = as.integer(1:10) # integer
for (i in c(NA, 1, 2, 4)) {
  y = base64encode(x, size=i)
  z = base64decode(y, typeof(x), size=i)
  stopifnot(x==z)
}
print("Checked base64 encode/decode for integer type")

x = (1:10)*pi        # double
for (i in c(NA, 4, 8)) {
  y = base64encode(x, size=i)
  z = base64decode(y, typeof(x), size=i)
  stopifnot(mean(abs(x-z))<1e-5)
}
print("Checked base64 for encode/decode double type")

x = log(as.complex(-(1:10)*pi))        # complex
```

```
y = base64encode(x)
z = base64decode(y, typeof(x))
stopifnot(x==z)
print("Checked base64 for encode/decode complex type")

x = "Chance favors the prepared mind" # character
y = base64encode(x)
z = base64decode(y, typeof(x))
stopifnot(x==z)
print("Checked base64 for encode/decode character type")
```

colAUC *Column-wise Area Under ROC Curve (AUC)*

Description

Calculate Area Under the ROC Curve (AUC) for every column of a matrix. Also, can be used to plot the ROC curves.

Usage

```
colAUC(X, y, plotROC=FALSE, alg=c("Wilcoxon","ROC"))
```

Arguments

X	A matrix or data frame. Rows contain samples and columns contain features/variables.
y	Class labels for the X data samples. A response vector with one label for each row/component of X. Can be either a factor, string or a numeric vector.
plotROC	Plot ROC curves. Use only for small number of features. If TRUE, will set alg to "ROC".
alg	Algorithm to use: "ROC" integrates ROC curves, while "Wilcoxon" uses Wilcoxon Rank Sum Test to get the same results. Default "Wilcoxon" is faster. This argument is mostly provided for verification.

Details

AUC is a very useful measure of similarity between two classes measuring area under "Receiver Operating Characteristic" or ROC curve. In case of data with no ties all sections of ROC curve are either horizontal or vertical, in case of data with ties diagonal sections can also occur. Area under the ROC curve is calculated using trapz function. AUC is always in between 0.5 (two classes are statistically identical) and 1.0 (there is a threshold value that can achieve a perfect separation between the classes).

Area under ROC Curve (AUC) measure is very similar to Wilcoxon Rank Sum Test (see wilcox.test) and Mann-Whitney U Test.

There are numerous other functions for calculating AUC in other packages. Unfortunately none of them had all the properties that were needed for classification preprocessing, to lower the dimensionality of the data (from tens of thousands to hundreds) before applying standard classification algorithms.

The main properties of this code are:

- Ability to work with multi-dimensional data (X can have many columns).
- Ability to work with multi-class datasets (y can have more than 2 different values).
- Speed - this code was written to calculate AUC's of large number of features, fast.
- Returned AUC is always bigger than 0.5, which is equivalent of testing for each feature colAUC(x,y) and colAUC(-x,y) and returning the value of the bigger one.

If those properties do not fit your problem, see "See Also" and "Examples" sections for AUC functions in other packages that might be a better fit for your needs.

Value

An output is a single matrix with the same number of columns as X and "n choose 2" ($\frac{n!}{(n-2)!2!} = n(n-1)/2$) number of rows, where n is number of unique labels in y list. For example, if y contains only two unique class labels (length(unique(lab))==2) than output matrix will have a single row containing AUC of each column. If more than two unique labels are present than AUC is calculated for every possible pairing of classes ("n choose 2" of them).

For multi-class AUC "Total AUC" as defined by Hand & Till (2001) can be calculated by colMeans(auc).

Author(s)

Jarek Tuszynski (SAIC) <jaroslaw.w.tuszynski@saic.com>

References

- Mason, S.J. and Graham, N.E. (1982) Areas beneath the relative operating characteristics (ROC) and relative operating levels (ROL) curves: Statistical significance and interpretation, Q. J. R. Meteorol. Soc. 30 291-303.
- Fawcett, Tom (2004) ROC Graphs: Notes and Practical Considerations for Researchers
- Hand, David and Till, Robert (2001) A Simple Generalization of the Area Under the ROC Curve for Multiple Class Classification Problems; Machine Learning 45(2): 171-186
- See http://www.medicine.mcgill.ca/epidemiology/hanley/software/ to find articles below:
  - Hanley and McNeil (1982), The Meaning and Use of the Area under a Receiver Operating Characteristic (ROC) Curve, Radiology 143: 29-36.
  - Hanley and McNeil (1983), A Method of Comparing the Areas under ROC curves derived from same cases, Radiology 148: 839-843.
  - McNeil and Hanley (1984), Statistical Approaches to the Analysis of ROC curves, Medical Decision Making 4(2): 136-149.

See Also

- wilcox.test and pwilcox
- wilcox.exact from **exactRankTests** package
- wilcox_test from **coin** package

- performance from **ROCR** package
- ROC from **Epi** package
- roc.area from **verification** package
- rcorr.cens from **Hmisc** package

Examples

```
# Load MASS library with "cats" data set that have following columns: sex, body
# weight, hart weight. Calculate how good weights are in predicting sex of cats.
# 2 classes; 2 features; 144 samples
library(MASS); data(cats);
colAUC(cats[,2:3], cats[,1], plotROC=TRUE)

# Load rpart library with "kyphosis" data set that records if kyphosis
# deformation was present after corrective surgery. Calculate how good age,
# number and position of vertebrae are in predicting successful operation.
# 2 classes; 3 features; 81 samples
library(rpart); data(kyphosis);
colAUC(kyphosis[,2:4], kyphosis[,1], plotROC=TRUE)

# Example of 3-class 4-feature 150-sample iris data
data(iris)
colAUC(iris[,-5], iris[,5], plotROC=TRUE)
cat("Total AUC: \n");
colMeans(colAUC(iris[,-5], iris[,5]))

# Test plots in case of data without column names
Iris = as.matrix(iris[,-5])
dim(Iris) = c(600,1)
dim(Iris) = c(150,4)
colAUC(Iris, iris[,5], plotROC=TRUE)

# Compare calAUC with other functions designed for similar purpose
auc = matrix(NA,12,3)
rownames(auc) = c("colAUC(alg='ROC')", "colAUC(alg='Wilcox')", "sum(rank)",
    "wilcox.test", "wilcox_test", "wilcox.exact", "roc.area", "AUC",
    "performance", "ROC", "auROC", "rcorr.cens")
colnames(auc) = c("AUC(x)", "AUC(-x)", "AUC(x+noise)")
X = cbind(cats[,2], -cats[,2], cats[,2]+rnorm(nrow(cats)) )
y = ifelse(cats[,1]=='F',0,1)
for (i in 1:3) {
  x = X[,i]
  x1 = x[y==1]; n1 = length(x1);                 # prepare input data ...
  x2 = x[y==0]; n2 = length(x2);                 # ... into required format
  data = data.frame(x=x,y=factor(y))
  auc[1,i] = colAUC(x, y, alg="ROC")
  auc[2,i] = colAUC(x, y, alg="Wilcox")
  r = rank(c(x1,x2))
  auc[3,i] = (sum(r[1:n1]) - n1*(n1+1)/2) / (n1*n2)
  auc[4,i] = wilcox.test(x1, x2, exact=0)$statistic / (n1*n2)
  ## Not run:
  if (require("coin"))
```

```
    auc[5,i] = statistic(wilcox_test(x~y, data=data)) / (n1*n2)
  if (require("exactRankTests"))
    auc[6,i] = wilcox.exact(x, y, exact=0)$statistic / (n1*n2)
  if (require("verification"))
    auc[7,i] = roc.area(y, x)$A.tilda
  if (require("ROC"))
    auc[8,i] = AUC(rocdemo.sca(y, x, dxrule.sca))
  if (require("ROCR"))
    auc[9,i] = performance(prediction( x, y),"auc")@y.values[[1]]
  if (require("Epi"))   auc[10,i] = ROC(x,y,grid=0)$AUC
  if (require("limma")) auc[11,i] = auROC(y, x)
  if (require("Hmisc")) auc[12,i] = rcorr.cens(x, y)[1]

## End(Not run)
}
print(auc)
stopifnot(auc[1, ]==auc[2, ])   # results of 2 alg's in colAUC must be the same
stopifnot(auc[1,1]==auc[3,1])   # compare with wilcox.test results

# time trials
x = matrix(runif(100*1000),100,1000)
y = (runif(100)>0.5)
system.time(colAUC(x,y,alg="ROC"   ))
system.time(colAUC(x,y,alg="Wilcox"))
```

combs *All Combinations of k Elements from Vector v*

Description

Finds all unordered combinations of k elements from vector v.

Usage

```
combs(v,k)
```

Arguments

v	Any numeric vector
k	Number of elements to choose from vector v. Integer smaller or equal than length of v.

Value

combs(v,k) (where v has length n) creates a matrix with $\frac{n!}{(n-k)!k!}$ (n choose k) rows and k columns containing all possible combinations of n elements taken k at a time.

Author(s)

Jarek Tuszynski (SAIC) <jaroslaw.w.tuszynski@saic.com>

See Also

I discovered recently that R packages already have two functions with similar capabilities: `combinations` from **gTools** package and Also similar to Matlab's `nchoosek` function (`http://www.mathworks.com/access/helpdesk/help/techdoc/ref/nchoosek.html`)

Examples

```
combs(2:5, 3) # display examples
combs(c("cats", "dogs", "mice"), 2)

a = combs(1:4, 2)
b = matrix( c(1,1,1,2,2,3,2,3,4,3,4,4), 6, 2)
stopifnot(a==b)
```

`LogitBoost` *LogitBoost Classification Algorithm*

Description

Train logitboost classification algorithm using decision stumps (one node decision trees) as weak learners.

Usage

```
LogitBoost(xlearn, ylearn, nIter=ncol(xlearn))
```

Arguments

`xlearn` A matrix or data frame with training data. Rows contain samples and columns contain features

`ylearn` Class labels for the training data samples. A response vector with one label for each row/component of `xlearn`. Can be either a factor, string or a numeric vector.

`nIter` An integer, describing the number of iterations for which boosting should be run, or number of decision stumps that will be used.

Details

The function was adapted from logitboost.R function written by Marcel Dettling. See references and "See Also" section. The code was modified in order to make it much faster for very large data sets. The speed-up was achieved by implementing a internal version of decision stump classifier instead of using calls to `rpart`. That way, some of the most time consuming operations were precomputed once, instead of performing them at each iteration. Another difference is that training and testing phases of the classification process were split into separate functions.

Value

An object of class "LogitBoost" including components:

`Stump` List of decision stumps (one node decision trees) used:

- column 1: feature numbers or each stump, or which column each stump operates on
- column 2: threshold to be used for that column
- column 3: bigger/smaller info: 1 means that if values in the column are above threshold than corresponding samples will be labeled as `lablist[1]`. Value "-1" means the opposite.

If there are more than two classes, than several "Stumps" will be `cbind`'ed

`lablist` names of each class

Author(s)

Jarek Tuszynski (SAIC) <`jaroslaw.w.tuszynski@saic.com`>

References

Dettling and Buhlmann (2002), *Boosting for Tumor Classification of Gene Expression Data*.

See Also

- `predict.LogitBoost` has prediction half of LogitBoost code
- `logitboost` function from **logitboost** library (not in CRAN or BioConductor is very similar but much slower on very large datasets. It also perform optional cross-validation.

Examples

```
data(iris)
Data  = iris[,-5]
Label = iris[, 5]

# basic interface
model = LogitBoost(Data, Label, nIter=20)
Lab   = predict(model, Data)
Prob  = predict(model, Data, type="raw")
t     = cbind(Lab, Prob)
t[1:10, ]

# two alternative call syntax
p=predict(model,Data)
q=predict.LogitBoost(model,Data)
pp=p[!is.na(p)]; qq=q[!is.na(q)]
stopifnot(pp == qq)

# accuracy increases with nIter (at least for train set)
table(predict(model, Data, nIter= 2), Label)
table(predict(model, Data, nIter=10), Label)
table(predict(model, Data),          Label)

# example of spliting the data into train and test set
mask = sample.split(Label)
model = LogitBoost(Data[mask,], Label[mask], nIter=10)
table(predict(model, Data[!mask,], nIter=2), Label[!mask])
table(predict(model, Data[!mask,]),          Label[!mask])
```

predict.LogitBoost *Prediction Based on LogitBoost Classification Algorithm*

Description

Prediction or Testing using logitboost classification algorithm

Usage

```
## S3 method for class 'LogitBoost'
predict(object, xtest, type = c("class", "raw"), nIter=NA, ...)
```

Arguments

object	An object of class "LogitBoost" see "Value" section of LogitBoost for details
xtest	A matrix or data frame with test data. Rows contain samples and columns contain features
type	See "Value" section
nIter	An optional integer, used to lower number of iterations (decision stumps) used in the decision making. If not provided than the number will be the same as the one provided in LogitBoost. If provided than the results will be the same as running LogitBoost with fewer iterations.
...	not used but needed for compatibility with generic predict method

Details

Logitboost algorithm relies on a voting scheme to make classifications. Many (nIter of them) week classifiers are applied to each sample and their findings are used as votes to make the final classification. The class with the most votes "wins". However, with this scheme it is common for two cases have a tie (the same number of votes), especially if number of iterations is even. In that case NA is returned, instead of a label.

Value

If type = "class" (default) label of the class with maximal probability is returned for each sample. If type = "raw", the a-posterior probabilities for each class are returned.

Author(s)

Jarek Tuszynski (SAIC) <jaroslaw.w.tuszynski@saic.com>

See Also

LogitBoost has training half of LogitBoost code

Examples

```
# See LogitBoost example
```

---

read.ENVI & write.ENVI

Read and Write Binary Data in ENVI Format

Description

Read and write binary data in ENVI format, which is supported by most GIS software.

Usage

```
read.ENVI(filename, headerfile=paste(filename, ".hdr", sep=""))
write.ENVI (X, filename, interleave = c("bsq", "bil", "bip"))
```

Arguments

X	data to be saved in ENVI file. Can be a matrix or 3D array.
filename	character string with name of the file (connection)
headerfile	optional character string with name of the header file
interleave	optional character string specifying interleave to be used

Details

ENVI binary files use a generalized raster data format that consists of two parts:

- binary file - flat binary file equivalent to memory dump, as produced by writeBin in R or fwrite in C/C++.
- header file - small text (ASCII) file containing the metadata associated with the binary file. This file can contain the following fields, followed by equal sign and a variable:
  - samples - number of columns
  - lines - number of rows
  - bands - number of bands (channels, planes)
  - data type - following types are supported:
    - 1 - 1-byte unsigned integer
    - 2 - 2-byte signed integer
    - 3 - 4-byte signed integer

    - ∗ 4 - 4-byte float
    - ∗ 5 - 8-byte double
    - ∗ 9 - 2x8-byte complex number made up from 2 doubles
    - ∗ 12 - 2-byte unsigned integer
  - header offset - number of bytes to skip before raster data starts in binary file.
  - interleave - Permutations of dimensions in binary data:
    - ∗ BSQ - Band Sequential (X[col,row,band])
    - ∗ BIL - Band Interleave by Line (X[col,band,row])
    - ∗ BIP - Band Interleave by Pixel (X[band,col,row])
  - byte order - the endian-ness of the saved data:
    - ∗ 0 - means little-endian byte order, format used on PC/Intel machines
    - ∗ 1 - means big-endian (aka IEEE, aka "network") byte order, format used on UNIX and Macintosh machines

Fields samples, lines, bands, data type are required, while header offset, interleave, byte order are optional. All of them are in form of integers except interleave which is a string.

This generic format allows reading of many raw file formats, including those with embedded header information. Also it is a handy binary format to exchange data between PC and UNIX/Mac machines, as well as different languages like: C, Fortran, Matlab, etc. Especially since header files are simple enough to edit by hand.

File type supported by most of GIS (geographic information system) software including: ENVI software, Freelook (free file viewer by ENVI), ArcGIS, etc.

Value

Function read.ENVI returns either a matrix or 3D array. Function write.ENVI does not return anything.

Author(s)

Jarek Tuszynski (SAIC) <jaroslaw.w.tuszynski@saic.com>

See Also

Displaying of images can be done through functions: graphics::image, fields::image.plot and fields:add.image or spatstat:plot.im.

ENVI files are practically C-style memory-dumps as performed by readBin and writeBin functions plus separate meta-data header file.

GIF file formats can also store 3D data (see read.gif and write.gif functions).

Packages related to GIS data: **shapefiles**, **maptools**, **sp**, **spdep**, **adehabitat**, **GRASS**, **PBSmapping**.

Examples

```
X = array(1:60, 3:5)
write.ENVI(X, "temp.nvi")
Y = read.ENVI("temp.nvi")
stopifnot(X == Y)
readLines("temp.nvi.hdr")

d = c(20,30,40)
X = array(runif(prod(d)), d)
write.ENVI(X, "temp.nvi", interleave="bil")
Y = read.ENVI("temp.nvi")
stopifnot(X == Y)
readLines("temp.nvi.hdr")

file.remove("temp.nvi")
file.remove("temp.nvi.hdr")
```

---

read.gif & write.gif *Read and Write Images in GIF format*

Description

Read and write files in GIF format. Files can contain single images or multiple frames. Multi-frame images are saved as animated GIF's.

Usage

```
read.gif(filename, frame=0, flip=FALSE, verbose=FALSE)
write.gif(image, filename, col="gray", scale=c("smart", "never", "always"),
    transparent=NULL, comment=NULL, delay=0, flip=FALSE, interlace=FALSE)
```

Arguments

`filename` Character string with name of the file. In case of `read.gif` URL's are also allowed.

`image` Data to be saved as GIF file. Can be a 2D matrix or 3D array. Allowed formats in order of preference:

- array of integers in [0:255] range - this is format required by GIF file, and unless `scale='always'`, numbers will not be rescaled. Each pixel `i` will have associated color `col[image[i]+1]`. This is the only format that can be safely used with non-continuous color maps.
- array of doubles in [0:1] range - Unless `scale='never'` the array will be multiplied by 255 and rounded.
- array of numbers in any range - will be scaled or clipped depending on `scale` option.

`frame` Request specific frame from multiframe (i.e., animated) GIF file. By default all frames are read from the file (`frame=0`). Setting `frame=1` will ensure that output is always a 2D matrix containing the first frame. Some files have to be read frame by frame, for example: files with subimages of different sizes and files with both global and local color-maps (palettes).

`col` Color palette definition. Several formats are allowed:

- array (list) of colors in the same format as output of palette functions. Preferred format for precise color control.
- palette function itself (ex. '`col=rainbow`'). Preferred format if not sure how many colors are needed.
- character string with name of internally defined palette. At the moment only "gray" and "jet" (Matlab's jet palette) are defined.
- character string with name of palette function (ex. '`col="rainbow"`')

Usually palette will consist of 256 colors, which is the maximum allowed by GIF format. By default, grayscale will be used.

`scale` There are three approaches to rescaling the data to required [0, 255] integer range:

- "smart" - Data is fitted to [0:255] range, only if it falls outside of it. Also, if `image` is an array of doubles in range [0, 1] than data is multiplied by 255.
- "never" - Pixels with intensities outside of the allowed range are clipped to either 0 or 255. Warning is given.
- "always" - Data is always rescaled. If `image` is a array of doubles in range [0, 1] than data is multiplied by 255; otherwise it is scaled to fit to [0:255] range.

`delay` In case of 3D arrays the data will be stored as animated GIF, and `delay` controls speed of the animation. It is number of hundredths (1/100) of a second of delay between frames.

`comment` Comments in text format are allowed in GIF files. Few file viewers can access them.

`flip` By default data is stored in the same orientation as data displayed by `print` function: row 1 is on top, image x-axis corresponds to columns and y-axis corresponds to rows. However function `image` adopted different standard: column 1 is on the bottom, image x-axis corresponds to rows and y-axis corresponds to columns. Set `flip` to `TRUE` to get the orientation used by `image`.

`transparent` Optional color number to be shown as transparent. Has to be an integer in [0:255] range. NA's in the `image` will be set to transparent.

`interlace` GIF files allow image rows to be `interlaced`, or reordered in such a way as to allow viewer to display image using 4 passes, making image sharper with each pass. Irrelevant feature on fast computers.

`verbose` Display details sections encountered while reading GIF file.

Details

Palettes often contain continuous colors, such that swapping palettes or rescaling of the image date does not affect image apperance in a drastic way. However, when working with non-continuous color-maps one should always provide image in [0:255] integer range (and set `scale="never"`), in order to prevent scaling.

If `NA` or other infinite numbers are found in the `image` by `write.gif`, they will be converted to numbers given by `transparent`. If `transparent` color is not provided than it will be created, possibly after reshretching.

There are some GIF files not fully supported by `read.gif` function:

- "Plain Text Extension" is not supported, and will be ignored.
- Multi-frame files with unique settings for each frame have to be read frame by frame. Possible settings include: frames with different sizes, frames using local color maps and frames using individual transparency colors.

Value

Function `write.gif` does not return anything. Function `read.gif` returns a list with following fields:

`image`	matrix or 3D array of integers in [0:255] range.
`col`	color palette definitions with number of colors ranging from 1 to 256. In case when `frame=0` only the first (usually global) color-map (palette) is returned.
`comment`	Comments imbedded in GIF File
`transparent`	color number corresponding to transparent color. If none was stated than NULL, otherwise an integer in [0:255] range. In order for `graphics:image` to display transparent colors correctly one should use `y$col[y$transparent+1] = NA`.

Author(s)

Jarek Tuszynski (SAIC) <`jaroslaw.w.tuszynski@saic.com`>. Encoding Algorithm adapted from code by Christoph Hohmann, which was adapted from code by Michael Mayer. Parts of decoding algorithm adapted from code by David Koblas.

References

Ziv, J., Lempel, A. (1977) *An Universal Algorithm for Sequential Data Compression*, IEEE Transactions on Information Theory, May 1977.

See Also

Displaying of images can be done through functions: `graphics:image` (part of R), `fields::image.plot` and `fields::add.image` or `spatstat:plot.im`, and possibly many other functions.

Displayed image can be saved in GIF, JPEG or PNG format using several different functions, like `R2HTML:HTMLplot`.

Functions for directly reading and writing image files:

- `read.pnm` and `pixmap::write.pnm` can process PBM, PGM and PPM images (file types supported by ImageMagic software)

- read.ENVI and write.ENVI from this package can process files in ENVI format. ENVI files can store 2D images and 3D data (multi-frame images), and are supported by most GIS (Geographic Information System) software including free "freelook".

There are many functions for creating and managing color palettes:

- fields::tim.colors contains a palette similar to Matlab's jet palette (see examples for simpler implementation)
- gplots::rich.colors contains two palettes of continuous colors.
- Functions RColorBrewer::brewer.pal and epitools::colorbrewer.palette contain tools for generating palettes.
- grDevices::rgb and grDevices::hsv create palette from RGB or HSV 3-vectors.
- grDevices::col2rgb translates palette colors to RGB 3-vectors.

Examples

```
# visual comparison between image and plot
write.gif( volcano, "volcano.gif", col=terrain.colors, flip=TRUE,
           scale="always", comment="Maunga Whau Volcano")
y = read.gif("volcano.gif", verbose=TRUE, flip=TRUE)
image(y$image, col=y$col, main=y$comment, asp=1)
# browseURL("file://volcano.gif")  # inspect GIF file on your hard disk

# test reading & writing
col = heat.colors(256) # choose colormap
trn = 222              # set transparent color
com = "Hello World"    # imbed comment in the file
write.gif( volcano, "volcano.gif", col=col, transparent=trn, comment=com)
y = read.gif("volcano.gif")
# This tested col==y$col, but colours may or may not have an alpha channel
# and for col this changed in R 4.0
stopifnot(volcano==y$image, trn==y$transparent, com==y$comment)
# browseURL("file://volcano.gif") # inspect GIF file on your hard disk

# create simple animated GIF (using image function in a loop is very rough,
# but only way I know of displaying 'animation" in R)
x <- y <- seq(-4*pi, 4*pi, len=200)
r <- sqrt(outer(x^2, y^2, "+"))
image = array(0, c(200, 200, 10))
for(i in 1:10) image[,,i] = cos(r-(2*pi*i/10))/(r^.25)
write.gif(image, "wave.gif", col="rainbow")
y = read.gif("wave.gif")
for(i in 1:10) image(y$image[,,i], col=y$col, breaks=(0:256)-0.5, asp=1)
# browseURL("file://wave.gif") # inspect GIF file on your hard disk

# Another neat animation of Mandelbrot Set
jet.colors = colorRampPalette(c("#00007F", "blue", "#007FFF", "cyan", "#7FFF7F",
             "yellow", "#FF7F00", "red", "#7F0000")) # define "jet" palette
m = 400
C = complex( real=rep(seq(-1.8,0.6, length.out=m), each=m ),
             imag=rep(seq(-1.2,1.2, length.out=m),      m ) )
```

```
C = matrix(C,m,m)
Z = 0
X = array(0, c(m,m,20))
for (k in 1:20) {
  Z = Z^2+C
  X[,,k] = exp(-abs(Z))
}
image(X[,,k], col=jet.colors(256))
write.gif(X, "Mandelbrot.gif", col=jet.colors, delay=100)
# browseURL("file://Mandelbrot.gif") # inspect GIF file on your hard disk
file.remove("wave.gif", "volcano.gif", "Mandelbrot.gif")

# Display interesting images from the web
## Not run:
url = "http://www.ngdc.noaa.gov/seg/cdroms/ged_iib/datasets/b12/gifs/eccnv.gif"
y = read.gif(url, verbose=TRUE, flip=TRUE)
image(y$image, col=y$col, breaks=(0:length(y$col))-0.5, asp=1,
           main="January Potential Evapotranspiration mm/mo")
url = "http://www.ngdc.noaa.gov/seg/cdroms/ged_iib/datasets/b01/gifs/fvvcode.gif"
y = read.gif(url, flip=TRUE)
y$col[y$transparent+1] = NA # mark transparent color in R way
image(y$image, col=y$col[1:87], breaks=(0:87)-0.5, asp=1,
           main="Vegetation Types")
url = "http://talc.geo.umn.edu/people/grads/hasba002/erosion_vids/run2/r2_dems_5fps(8color).gif"
y = read.gif(url, verbose=TRUE, flip=TRUE)
for(i in 2:dim(y$image)[3])
  image(y$image[,,i], col=y$col, breaks=(0:length(y$col))-0.5,
            asp=1, main="Erosion in Drainage Basins")

## End(Not run)
```

runmad *Median Absolute Deviation of Moving Windows*

Description

Moving (aka running, rolling) Window MAD (Median Absolute Deviation) calculated over a vector

Usage

```
runmad(x, k, center = runmed(x,k), constant = 1.4826,
      endrule=c("mad", "NA", "trim", "keep", "constant", "func"),
      align = c("center", "left", "right"))
```

Arguments

x numeric vector of length n or matrix with n rows. If x is a matrix than each column will be processed separately.

`k` width of moving window; must be an integer between one and n. In case of even k's one will have to provide different `center` function, since `runmed` does not take even k's.

`endrule` character string indicating how the values at the beginning and the end, of the data, should be treated. Only first and last `k2` values at both ends are affected, where `k2` is the half-bandwidth `k2 = k %/% 2`.

- `"mad"` - applies the mad function to smaller and smaller sections of the array. Equivalent to: `for(i in 1:k2) out[i]=mad(x[1:(i+k2)])`.
- `"trim"` - trim the ends; output array length is equal to `length(x)-2*k2` (`out = out[(k2+1):(n-k2)]`). This option mimics output of `apply (embed(x,k),1,FUN)` and other related functions.
- `"keep"` - fill the ends with numbers from `x` vector (`out[1:k2] = x[1:k2]`). This option makes more sense in case of smoothing functions, kept here for consistency.
- `"constant"` - fill the ends with first and last calculated value in output array (`out[1:k2] = out[k2+1]`)
- `"NA"` - fill the ends with NA's (`out[1:k2] = NA`)
- `"func"` - same as `"mad"` option except that implemented in R for testing purposes. Avoid since it can be very slow for large windows.

Similar to `endrule` in `runmed` function which has the following options: "`c("median", "keep", "constant")`".

`center` moving window center. Defaults to running median (`runmed` function). Similar to `center` in `mad` function. For best acuracy at the edges use `runquantile(x,k,0.5,type=2)`, which is slower than default `runmed(x,k,endrule="med")`. If `x` is a 2D array (and `endrule="mad"`) or if `endrule="func"` than array edges are filled by repeated calls to "`mad(x, center=median(x), na.rm=TRUE)`" function. Runmad's `center` parameter will be ignored for the beggining and the end of output `y`. Please use `center=runquantile(x,k,0.5,type=2)` for those cases.

`constant` scale factor such that for Gaussian distribution X, `mad`(X) is the same as `sd`(X). Same as `constant` in `mad` function.

`align` specifies whether result should be centered (default), left-aligned or right-aligned. If `endrule`="mad" then setting `align` to "left" or "right" will fall back on slower implementation equivalent to `endrule`="func".

Details

Apart from the end values, the result of y = runmad(x, k) is the same as "`for(j=(1+k2):(n-k2)) y[j]=mad(x[(j-k2):(j+k2)], na.rm = TRUE)`". It can handle non-finite numbers like NaN's and Inf's (like "`mad(x, na.rm = TRUE)`").

The main incentive to write this set of functions was relative slowness of majority of moving window functions available in R and its packages. With the exception of `runmed`, a running window median function, all functions listed in "see also" section are slower than very inefficient "`apply(embed(x,k),1,FUN)`" approach.

Functions `runquantile` and `runmad` are using insertion sort to sort the moving window, but gain speed by remembering results of the previous sort. Since each time the window is moved, only one point changes, all but one points in the window are already sorted. Insertion sort can fix that in O(k) time.

Value

Returns a numeric vector or matrix of the same size as x. Only in case of endrule="trim" the output vectors will be shorter and output matrices will have fewer rows.

Author(s)

Jarek Tuszynski (SAIC) <jaroslaw.w.tuszynski@saic.com>

References

About insertion sort used in runmad function see: R. Sedgewick (1988): *Algorithms*. Addison-Wesley (page 99)

See Also

Links related to:

- runmad - mad
- Other moving window functions from this package: runmin, runmax, runquantile, runmean and runsd
- generic running window functions: apply (embed(x,k), 1, FUN) (fastest), running from **gtools** package (extremely slow for this purpose), subsums from **magic** library can perform running window operations on data with any dimensions.

Examples

```
# show runmed function
k=25; n=200;
x = rnorm(n,sd=30) + abs(seq(n)-n/4)
col = c("black", "red", "green")
m=runmed(x, k)
y=runmad(x, k, center=m)
plot(x, col=col[1], main = "Moving Window Analysis Functions")
lines(m    , col=col[2])
lines(m-y/2, col=col[3])
lines(m+y/2, col=col[3])
lab = c("data", "runmed", "runmed-runmad/2", "runmed+runmad/2")
legend(0,0.9*n, lab, col=col, lty=1 )

# basic tests against apply/embed
eps = .Machine$double.eps ^ 0.5
k=25 # odd size window
a = runmad(x,k, center=runmed(x,k), endrule="trim")
b = apply(embed(x,k), 1, mad)
stopifnot(all(abs(a-b)<eps));
k=24 # even size window
a = runmad(x,k, center=runquantile(x,k,0.5,type=2), endrule="trim")
b = apply(embed(x,k), 1, mad)
stopifnot(all(abs(a-b)<eps));

# test against loop approach
```

```
  # this test works fine at the R prompt but fails during package check - need to investigate
  k=24; n=200;
  x = rnorm(n,sd=30) + abs(seq(n)-n/4) # create random data
  x = rep(1:5,40)
  #x[seq(1,n,11)] = NaN;               # commented out for time beeing - on to do list
  #x[5] = NaN;                         # commented out for time beeing - on to do list
  k2 = k
  k1 = k-k2-1
  ac = array(runquantile(x,k,0.5))
  a  = runmad(x, k, center=ac)
  bc = array(0,n)
  b  = array(0,n)
  for(j in 1:n) {
    lo = max(1, j-k1)
    hi = min(n, j+k2)
    bc[j] = median(x[lo:hi], na.rm = TRUE)
    b [j] = mad   (x[lo:hi], na.rm = TRUE, center=bc[j])
  }
  eps = .Machine$double.eps ^ 0.5
  #stopifnot(all(abs(ac-bc)<eps)); # commented out for time beeing - on to do list
  #stopifnot(all(abs(a-b)<eps));   # commented out for time beeing - on to do list

  # compare calculation at array ends
  k=25; n=200;
  x = rnorm(n,sd=30) + abs(seq(n)-n/4)
  c = runquantile(x,k,0.5,type=2)             # find the center
  a = runmad(x, k, center=c, endrule=”mad” )  # fast C code
  b = runmad(x, k, center=c, endrule=”func”)  # slow R code
  stopifnot(all(abs(a-b)<eps));

  # test if moving windows forward and backward gives the same results
  k=51;
  a = runmad(x     , k)
  b = runmad(x[n:1], k)
  stopifnot(all(a[n:1]==b, na.rm=TRUE));

  # test vector vs. matrix inputs, especially for the edge handling
  nRow=200; k=25; nCol=10
  x = rnorm(nRow,sd=30) + abs(seq(nRow)-n/4)
  X = matrix(rep(x, nCol ), nRow, nCol) # replicate x in columns of X
  a = runmad(x, k, center = runquantile(x,k,0.5,type=2))
  b = runmad(X, k, center = runquantile(X,k,0.5,type=2))
  stopifnot(all(abs(a-b[,1])<eps));        # vector vs. 2D array
  stopifnot(all(abs(b[,1]-b[,nCol])<eps)); # compare rows within 2D array

  # speed comparison
  ## Not run:
  x=runif(1e5); k=51;                       # reduce vector and window sizes
  system.time(runmad( x,k,endrule=”trim”))
  system.time(apply(embed(x,k), 1, mad))

## End(Not run)
```

runmean *Mean of a Moving Window*

Description

Moving (aka running, rolling) Window Mean calculated over a vector

Usage

```
runmean(x, k, alg=c("C", "R", "fast", "exact"),
       endrule=c("mean", "NA", "trim", "keep", "constant", "func"),
       align = c("center", "left", "right"))
```

Arguments

x numeric vector of length n or matrix with n rows. If x is a matrix than each column will be processed separately.

k width of moving window; must be an integer between 1 and n

alg an option to choose different algorithms

- "C" - a version is written in C. It can handle non-finite numbers like NaN's and Inf's (like mean(x, na.rm = TRUE)). It works the fastest for endrule="mean".
- "fast" - second, even faster, C version. This algorithm does not work with non-finite numbers. It also works the fastest for endrule other than "mean".
- "R" - much slower code written in R. Useful for debugging and as documentation.
- "exact" - same as "C", except that all additions are performed using algorithm which tracks and corrects addition round-off errors

endrule character string indicating how the values at the beginning and the end, of the data, should be treated. Only first and last k2 values at both ends are affected, where k2 is the half-bandwidth k2 = k %/% 2.

- "mean" - applies the underlying function to smaller and smaller sections of the array. Equivalent to: for(i in 1:k2) out[i] = mean(x[1:(i+k2)]). This option is implemented in C if alg="C", otherwise is done in R.
- "trim" - trim the ends; output array length is equal to length(x)-2*k2 (out = out[(k2+1):(n-k2)]). This option mimics output of apply (embed(x,k),1,mean) and other related functions.
- "keep" - fill the ends with numbers from x vector (out[1:k2] = x[1:k2])
- "constant" - fill the ends with first and last calculated value in output array (out[1:k2] = out[k2+1])
- "NA" - fill the ends with NA's (out[1:k2] = NA)
- "func" - same as "mean" but implimented in R. This option could be very slow, and is included mostly for testing

Similar to `endrule` in `runmed` function which has the following options: “`c("median", "keep", "constant")`”.

`align` specifies whether result should be centered (default), left-aligned or right-aligned. If `endrule`="mean" then setting `align` to "left" or "right" will fall back on slower implementation equivalent to `endrule`="func".

Details

Apart from the end values, the result of y = runmean(x, k) is the same as “`for(j=(1+k2):(n-k2)) y[j]=mean(x[(j-k2):(j+k2)])`”.

The main incentive to write this set of functions was relative slowness of majority of moving window functions available in R and its packages. With the exception of `runmed`, a running window median function, all functions listed in "see also" section are slower than very inefficient “`apply(embed(x,k),1,FUN)`” approach. Relative speed of `runmean` function is O(n).

Function `EndRule` applies one of the five methods (see `endrule` argument) to process end-points of the input array `x`. In current version of the code the default `endrule="mean"` option is calculated within C code. That is done to improve speed in case of large moving windows.

In case of `runmean(..., alg="exact")` function a special algorithm is used (see references section) to ensure that round-off errors do not accumulate. As a result `runmean` is more accurate than `filter`(x, rep(1/k,k)) and `runmean(..., alg="C")` functions.

Value

Returns a numeric vector or matrix of the same size as `x`. Only in case of `endrule="trim"` the output vectors will be shorter and output matrices will have fewer rows.

Note

Function `runmean(..., alg="exact")` is based by code by Vadim Ogranovich, which is based on Python code (see last reference), pointed out by Gabor Grothendieck.

Author(s)

Jarek Tuszynski (SAIC) <`jaroslaw.w.tuszynski@saic.com`>

References

- About round-off error correction used in `runmean`: Shewchuk, Jonathan *Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates*, `http://www-2.cs.cmu.edu/afs/cs/project/quake/public/papers/robust-arithmetic.ps`

See Also

Links related to:

- moving mean - `mean`, `kernapply`, `filter`, `decompose`, `stl`, `rollmean` from **zoo** library, `subsums` from **magic** library,
- Other moving window functions from this package: `runmin`, `runmax`, `runquantile`, `runmad` and `runsd`

- runmed
- generic running window functions: apply (embed(x,k), 1, FUN) (fastest), running from **gtools** package (extremely slow for this purpose), subsums from **magic** library can perform running window operations on data with any dimensions.

Examples

```
# show runmean for different window sizes
n=200;
x = rnorm(n,sd=30) + abs(seq(n)-n/4)
x[seq(1,n,10)] = NaN;              # add NANs
col = c("black", "red", "green", "blue", "magenta", "cyan")
plot(x, col=col[1], main = "Moving Window Means")
lines(runmean(x, 3), col=col[2])
lines(runmean(x, 8), col=col[3])
lines(runmean(x,15), col=col[4])
lines(runmean(x,24), col=col[5])
lines(runmean(x,50), col=col[6])
lab = c("data", "k=3", "k=8", "k=15", "k=24", "k=50")
legend(0,0.9*n, lab, col=col, lty=1 )

# basic tests against 2 standard R approaches
k=25; n=200;
x = rnorm(n,sd=30) + abs(seq(n)-n/4)      # create random data
a = runmean(x,k, endrule="trim")          # tested function
b = apply(embed(x,k), 1, mean)            # approach #1
c = cumsum(c( sum(x[1:k]), diff(x,k) ))/k # approach #2
eps = .Machine$double.eps ^ 0.5
stopifnot(all(abs(a-b)<eps));
stopifnot(all(abs(a-c)<eps));

# test against loop approach
# this test works fine at the R prompt but fails during package check - need to investigate
k=25;
data(iris)
x = iris[,1]
n = length(x)
x[seq(1,n,11)] = NaN;                # add NANs
k2 = k
k1 = k-k2-1
a = runmean(x, k)
b = array(0,n)
for(j in 1:n) {
  lo = max(1, j-k1)
  hi = min(n, j+k2)
  b[j] = mean(x[lo:hi], na.rm = TRUE)
}
#stopifnot(all(abs(a-b)<eps)); # commented out for time beeing - on to do list

# compare calculation at array ends
a = runmean(x, k, endrule="mean")  # fast C code
b = runmean(x, k, endrule="func")  # slow R code
```

```
stopifnot(all(abs(a-b)<eps));

# Testing of different methods to each other for non-finite data
# Only alg "C" and "exact" can handle not finite numbers
eps = .Machine$double.eps ^ 0.5
n=200;  k=51;
x = rnorm(n,sd=30) + abs(seq(n)-n/4) # nice behaving data
x[seq(1,n,10)] = NaN;                # add NANs
x[seq(1,n, 9)] = Inf;                # add infinities
b = runmean( x, k, alg="C")
c = runmean( x, k, alg="exact")
stopifnot(all(abs(b-c)<eps));

# Test if moving windows forward and backward gives the same results
# Test also performed on data with non-finite numbers
a = runmean(x     , alg="C", k)
b = runmean(x[n:1], alg="C", k)
stopifnot(all(abs(a[n:1]-b)<eps));
a = runmean(x     , alg="exact", k)
b = runmean(x[n:1], alg="exact", k)
stopifnot(all(abs(a[n:1]-b)<eps));

# test vector vs. matrix inputs, especially for the edge handling
nRow=200; k=25; nCol=10
x = rnorm(nRow,sd=30) + abs(seq(nRow)-n/4)
x[seq(1,nRow,10)] = NaN;              # add NANs
X = matrix(rep(x, nCol ), nRow, nCol) # replicate x in columns of X
a = runmean(x, k)
b = runmean(X, k)
stopifnot(all(abs(a-b[,1])<eps));        # vector vs. 2D array
stopifnot(all(abs(b[,1]-b[,nCol])<eps)); # compare rows within 2D array

# Exhaustive testing of different methods to each other for different windows
numeric.test = function (x, k) {
  a = runmean( x, k, alg="fast")
  b = runmean( x, k, alg="C")
  c = runmean( x, k, alg="exact")
  d = runmean( x, k, alg="R", endrule="func")
  eps = .Machine$double.eps ^ 0.5
  stopifnot(all(abs(a-b)<eps));
  stopifnot(all(abs(b-c)<eps));
  stopifnot(all(abs(c-d)<eps));
}
n=200;
x = rnorm(n,sd=30) + abs(seq(n)-n/4) # nice behaving data
for(i in 1:5) numeric.test(x, i)     # test small window sizes
for(i in 1:5) numeric.test(x, n-i+1) # test large window size

# speed comparison
## Not run:
x=runif(1e7); k=1e4;
system.time(runmean(x,k,alg="fast"))
system.time(runmean(x,k,alg="C"))
```

```
  system.time(runmean(x,k,alg="exact"))
  system.time(runmean(x,k,alg="R"))           # R version of the function
  x=runif(1e5); k=1e2;                        # reduce vector and window sizes
  system.time(runmean(x,k,alg="R"))           # R version of the function
  system.time(apply(embed(x,k), 1, mean))     # standard R approach
  system.time(filter(x, rep(1/k,k), sides=2)) # the fastest alternative I know

## End(Not run)

  # show different runmean algorithms with data spanning many orders of magnitude
  n=30; k=5;
  x = rep(100/3,n)
  d=1e10
  x[5] = d;
  x[13] = d;
  x[14] = d*d;
  x[15] = d*d*d;
  x[16] = d*d*d*d;
  x[17] = d*d*d*d*d;
  a = runmean(x, k, alg="fast" )
  b = runmean(x, k, alg="C"    )
  c = runmean(x, k, alg="exact")
  y = t(rbind(x,a,b,c))
  y
```

runmin & runmax *Minimum and Maximum of Moving Windows*

Description

Moving (aka running, rolling) Window Minimum and Maximum calculated over a vector

Usage

```
runmin(x, k, alg=c("C", "R"),
       endrule=c("min", "NA", "trim", "keep", "constant", "func"),
       align = c("center", "left", "right"))
runmax(x, k, alg=c("C", "R"),
       endrule=c("max", "NA", "trim", "keep", "constant", "func"),
       align = c("center", "left", "right"))
```

Arguments

`x` numeric vector of length n or matrix with n rows. If x is a matrix than each column will be processed separately.

`k` width of moving window; must be an integer between one and n

`endrule` character string indicating how the values at the beginning and the end, of the array, should be treated. Only first and last `k2` values at both ends are affected, where `k2` is the half-bandwidth `k2 = k %/% 2`.

- `"min"` & `"max"` - applies the underlying function to smaller and smaller sections of the array. In case of min equivalent to: `for(i in 1:k2) out[i]=min(x[1:(i+k2)])`. Default.
- `"trim"` - trim the ends; output array length is equal to `length(x)-2*k2` `(out = out[(k2+1):(n-k2)])`. This option mimics output of `apply (embed(x,k),1,FUN)` and other related functions.
- `"keep"` - fill the ends with numbers from `x` vector `(out[1:k2] = x[1:k2])`
- `"constant"` - fill the ends with first and last calculated value in output array `(out[1:k2] = out[k2+1])`
- `"NA"` - fill the ends with NA's `(out[1:k2] = NA)`
- `"func"` - same as `"min"` & `"max"` but implimented in R. This option could be very slow, and is included mostly for testing

Similar to `endrule` in `runmed` function which has the following options: “`c("median", "keep", "constant")`” .

`alg` an option allowing to choose different algorithms or implementations. Default is to use of code written in C (option `alg="C"`). Option `alg="R"` will use slower code written in R. Useful for debugging and studying the algorithm.

`align` specifies whether result should be centered (default), left-aligned or right-aligned. If `endrule`="min" or "max" then setting `align` to "left" or "right" will fall back on slower implementation equivalent to `endrule`="func".

Details

Apart from the end values, the result of y = runFUN(x, k) is the same as “`for(j=(1+k2):(n-k2)) y[j]=FUN(x[(j-k2):(j+k2)], na.rm = TRUE)`”, where FUN stands for min or max functions. Both functions can handle non-finite numbers like NaN's and Inf's the same way as `min(x, na.rm = TRUE))`.

The main incentive to write this set of functions was relative slowness of majority of moving window functions available in R and its packages. With the exception of `runmed`, a running window median function, all functions listed in "see also" section are slower than very inefficient “`apply(embed(x,k),1,FUN)`” approach. Relative speeds `runmin` and `runmax` functions is O(n) in best and average case and O(n*k) in worst case.

Both functions work with infinite numbers (`NA`,`NaN`,`Inf`, `-Inf`). Also default `endrule` is hardwired in C for speed.

Value

Returns a numeric vector or matrix of the same size as `x`. Only in case of `endrule="trim"` the output vectors will be shorter and output matrices will have fewer rows.

Author(s)

Jarek Tuszynski (SAIC) `<jaroslaw.w.tuszynski@saic.com>`

See Also

Links related to:

- Other moving window functions from this package: runmean, runquantile, runmad and runsd
- R functions: runmed, min, max
- Similar functions in other packages: rollmax from **zoo** library
- generic running window functions: apply (embed(x,k), 1, FUN) (fastest), running from **gtools** package (extremely slow for this purpose), subsums from **magic** library can perform running window operations on data with any dimensions.

Examples

```
# show plot using runmin, runmax and runmed
k=25; n=200;
x = rnorm(n,sd=30) + abs(seq(n)-n/4)
col = c("black", "red", "green", "blue", "magenta", "cyan")
plot(x, col=col[1], main = "Moving Window Analysis Functions")
lines(runmin(x,k), col=col[2])
lines(runmean(x,k), col=col[3])
lines(runmax(x,k), col=col[4])
legend(0,.9*n, c("data", "runmin", "runmean", "runmax"), col=col, lty=1 )

# basic tests against standard R approach
a = runmin(x,k, endrule="trim") # test only the inner part
b = apply(embed(x,k), 1, min)   # Standard R running min
stopifnot(all(a==b));
a = runmax(x,k, endrule="trim") # test only the inner part
b = apply(embed(x,k), 1, max)   # Standard R running min
stopifnot(all(a==b));

# test against loop approach
k=25;
data(iris)
x = iris[,1]
n = length(x)
x[seq(1,n,11)] = NaN;                # add NANs
k2 = k
k1 = k-k2-1
a1 = runmin(x, k)
a2 = runmax(x, k)
b1 = array(0,n)
b2 = array(0,n)
for(j in 1:n) {
  lo = max(1, j-k1)
  hi = min(n, j+k2)
  b1[j] = min(x[lo:hi], na.rm = TRUE)
  b2[j] = max(x[lo:hi], na.rm = TRUE)
}
# this test works fine at the R prompt but fails during package check - need to investigate
## Not run:
  stopifnot(all(a1==b1, na.rm=TRUE));
  stopifnot(all(a2==b2, na.rm=TRUE));

## End(Not run)

  # Test if moving windows forward and backward gives the same results
  # Two data sets also corespond to best and worst-case scenatio data-sets
  k=51; n=200;
  a = runmin(n:1, k)
  b = runmin(1:n, k)
  stopifnot(all(a[n:1]==b, na.rm=TRUE));
  a = runmax(n:1, k)
  b = runmax(1:n, k)
  stopifnot(all(a[n:1]==b, na.rm=TRUE));

  # test vector vs. matrix inputs, especially for the edge handling
  nRow=200; k=25; nCol=10
  x = rnorm(nRow,sd=30) + abs(seq(nRow)-n/4)
  x[seq(1,nRow,10)] = NaN;              # add NANs
  X = matrix(rep(x, nCol ), nRow, nCol) # replicate x in columns of X
  a = runmax(x, k)
  b = runmax(X, k)
  stopifnot(all(a==b[,1], na.rm=TRUE));        # vector vs. 2D array
  stopifnot(all(b[,1]==b[,nCol], na.rm=TRUE)); # compare rows within 2D array
  a = runmin(x, k)
  b = runmin(X, k)
  stopifnot(all(a==b[,1], na.rm=TRUE));        # vector vs. 2D array
  stopifnot(all(b[,1]==b[,nCol], na.rm=TRUE)); # compare rows within 2D array

  # Compare C and R algorithms to each other for extreme window sizes
  numeric.test = function (x, k) {
    a = runmin( x, k, alg="C")
    b = runmin( x, k, alg="R")
    c =-runmax(-x, k, alg="C")
    d =-runmax(-x, k, alg="R")
    stopifnot(all(a==b, na.rm=TRUE));
    #stopifnot(all(c==d, na.rm=TRUE));
    #stopifnot(all(a==c, na.rm=TRUE));
    stopifnot(all(b==d, na.rm=TRUE));
  }
  n=200;                             # n is an even number
  x = rnorm(n,sd=30) + abs(seq(n)-n/4) # random data
  for(i in 1:5) numeric.test(x, i)     # test for small window size
  for(i in 1:5) numeric.test(x, n-i+1) # test for large window size
  n=201;                             # n is an odd number
  x = rnorm(n,sd=30) + abs(seq(n)-n/4) # random data
  for(i in 1:5) numeric.test(x, i)     # test for small window size
  for(i in 1:5) numeric.test(x, n-i+1) # test for large window size
  n=200;                             # n is an even number
  x = rnorm(n,sd=30) + abs(seq(n)-n/4) # random data
  x[seq(1,200,10)] = NaN;            # with some NaNs
  for(i in 1:5) numeric.test(x, i)     # test for small window size
  for(i in 1:5) numeric.test(x, n-i+1) # test for large window size
  n=201;                               # n is an odd number
  x = rnorm(n,sd=30) + abs(seq(n)-n/4) # random data
  x[seq(1,200,2)] = NaN;               # with some NaNs
  for(i in 1:5) numeric.test(x, i)     # test for small window size
  for(i in 1:5) numeric.test(x, n-i+1) # test for large window size

  # speed comparison
  ## Not run:
  n = 1e7;  k=991;
  x1 = runif(n);                       # random data - average case scenario
  x2 = 1:n;                            #  best-case scenario data for runmax
  x3 = n:1;                            # worst-case scenario data for runmax
  system.time( runmax( x1,k,alg="C"))  # C alg on average data O(n)
  system.time( runmax( x2,k,alg="C"))  # C alg on  best-case data O(n)
  system.time( runmax( x3,k,alg="C"))  # C alg on worst-case data O(n*k)
  system.time(-runmin(-x1,k,alg="C"))  # use runmin to do runmax work
  system.time( runmax( x1,k,alg="R"))  # R version of the function
  x=runif(1e5); k=1e2;                 # reduce vector and window sizes
  system.time(runmax(x,k,alg="R"))     # R version of the function
  system.time(apply(embed(x,k), 1, max)) # standard R approach

## End(Not run)
```

runquantile *Quantile of Moving Window*

Description

Moving (aka running, rolling) Window Quantile calculated over a vector

Usage

```
runquantile(x, k, probs, type=7,
        endrule=c("quantile", "NA", "trim", "keep", "constant", "func"),
        align = c("center", "left", "right"))
```

Arguments

x — numeric vector of length n or matrix with n rows. If x is a matrix than each column will be processed separately.

k — width of moving window; must be an integer between one and n

endrule — character string indicating how the values at the beginning and the end, of the array, should be treated. Only first and last k2 values at both ends are affected, where k2 is the half-bandwidth k2 = k %/% 2.

- "quantile" - applies the quantile function to smaller and smaller sections of the array. Equivalent to: for(i in 1:k2) out[i]=quantile(x[1:(i+k2)]).
- "trim" - trim the ends; output array length is equal to length(x)-2*k2 (out = out[(k2+1):(n-k2)]). This option mimics output of apply (embed(x,k),1,FUN) and other related functions.

- "keep" - fill the ends with numbers from x vector (out[1:k2] = x[1:k2])
- "constant" - fill the ends with first and last calculated value in output array (out[1:k2] = out[k2+1])
- "NA" - fill the ends with NA's (out[1:k2] = NA)
- "func" - same as "quantile" but implimented in R. This option could be very slow, and is included mostly for testing

Similar to endrule in runmed function which has the following options: “c("median", "keep", "constant")” .

probs numeric vector of probabilities with values in [0,1] range used by runquantile. For example Probs=c(0,0.5,1) would be equivalent to running runmin, runmed and runmax. Same as probs in quantile function.

type an integer between 1 and 9 selecting one of the nine quantile algorithms, same as type in quantile function. Another even more readable description of nine ways to calculate quantiles can be found at http://mathworld.wolfram.com/Quantile.html.

align specifies whether result should be centered (default), left-aligned or right-aligned. If endrule="quantile" then setting align to "left" or "right" will fall back on slower implementation equivalent to endrule="func".

Details

Apart from the end values, the result of y = runquantile(x, k) is the same as “for(j=(1+k2):(n-k2)) y[j]=quintile(x[(j-k2):(j+k2)],na.rm = TRUE)”. It can handle non-finite numbers like NaN's and Inf's (like quantile(x, na.rm = TRUE)).

The main incentive to write this set of functions was relative slowness of majority of moving window functions available in R and its packages. With the exception of runmed, a running window median function, all functions listed in "see also" section are slower than very inefficient “apply(embed(x,k),1,FUN)” approach. Relative speeds of runquantile is O(n*k)

Functions runquantile and runmad are using insertion sort to sort the moving window, but gain speed by remembering results of the previous sort. Since each time the window is moved, only one point changes, all but one points in the window are already sorted. Insertion sort can fix that in O(k) time.

Value

If x is a matrix than function runquantile returns a matrix of size [n × length(probs)]. If x is vactor a than function runquantile returns a matrix of size [dim(x) × length(probs)]. If endrule="trim" the output will have fewer rows.

Author(s)

Jarek Tuszynski (SAIC) <jaroslaw.w.tuszynski@saic.com>

References

- About quantiles: Hyndman, R. J. and Fan, Y. (1996) *Sample quantiles in statistical packages, American Statistician*, 50, 361.

- About quantiles: Eric W. Weisstein. *Quantile*. From MathWorld– A Wolfram Web Resource. http://mathworld.wolfram.com/Quantile.html
- About insertion sort used in runmad and runquantile: R. Sedgewick (1988): *Algorithms*. Addison-Wesley (page 99)

See Also

Links related to:

- Running Quantile - quantile, runmed, smooth, rollmedian from **zoo** library
- Other moving window functions from this package: runmin, runmax, runmean, runmad and runsd
- Running Minimum - min
- Running Maximum - max, rollmax from **zoo** library
- generic running window functions: apply (embed(x,k), 1, FUN) (fastest), running from **gtools** package (extremely slow for this purpose), subsums from **magic** library can perform running window operations on data with any dimensions.

Examples

```
# show plot using runquantile
k=31; n=200;
x = rnorm(n,sd=30) + abs(seq(n)-n/4)
y=runquantile(x, k, probs=c(0.05, 0.25, 0.5, 0.75, 0.95))
col = c("black", "red", "green", "blue", "magenta", "cyan")
plot(x, col=col[1], main = "Moving Window Quantiles")
lines(y[,1], col=col[2])
lines(y[,2], col=col[3])
lines(y[,3], col=col[4])
lines(y[,4], col=col[5])
lines(y[,5], col=col[6])
lab = c("data", "runquantile(.05)", "runquantile(.25)", "runquantile(0.5)",
        "runquantile(.75)", "runquantile(.95)")
legend(0,230, lab, col=col, lty=1 )

# show plot using runquantile
k=15; n=200;
x = rnorm(n,sd=30) + abs(seq(n)-n/4)
y=runquantile(x, k, probs=c(0.05, 0.25, 0.5, 0.75, 0.95))
col = c("black", "red", "green", "blue", "magenta", "cyan")
plot(x, col=col[1], main = "Moving Window Quantiles (smoothed)")
lines(runmean(y[,1],k), col=col[2])
lines(runmean(y[,2],k), col=col[3])
lines(runmean(y[,3],k), col=col[4])
lines(runmean(y[,4],k), col=col[5])
lines(runmean(y[,5],k), col=col[6])
lab = c("data", "runquantile(.05)", "runquantile(.25)", "runquantile(0.5)",
        "runquantile(.75)", "runquantile(.95)")
legend(0,230, lab, col=col, lty=1 )
```

```
# basic tests against runmin & runmax
y = runquantile(x, k, probs=c(0, 1))
a = runmin(x,k) # test only the inner part
stopifnot(all(a==y[,1], na.rm=TRUE));
a = runmax(x,k) # test only the inner part
stopifnot(all(a==y[,2], na.rm=TRUE));

# basic tests against runmed, including testing endrules
a = runquantile(x, k, probs=0.5, endrule="keep")
b = runmed(x, k, endrule="keep")
stopifnot(all(a==b, na.rm=TRUE));
a = runquantile(x, k, probs=0.5, endrule="constant")
b = runmed(x, k, endrule="constant")
stopifnot(all(a==b, na.rm=TRUE));

# basic tests against apply/embed
a = runquantile(x,k, c(0.3, 0.7), endrule="trim")
b = t(apply(embed(x,k), 1, quantile, probs = c(0.3, 0.7)))
eps = .Machine$double.eps ^ 0.5
stopifnot(all(abs(a-b)<eps));

# test against loop approach
# this test works fine at the R prompt but fails during package check - need to investigate
k=25; n=200;
x = rnorm(n,sd=30) + abs(seq(n)-n/4) # create random data
x[seq(1,n,11)] = NaN;                # add NANs
k2 = k
k1 = k-k2-1
a = runquantile(x, k, probs=c(0.3, 0.8) )
b = matrix(0,n,2);
for(j in 1:n) {
  lo = max(1, j-k1)
  hi = min(n, j+k2)
  b[j,] = quantile(x[lo:hi], probs=c(0.3, 0.8), na.rm = TRUE)
}
#stopifnot(all(abs(a-b)<eps));

# compare calculation of array ends
a = runquantile(x, k, probs=0.4, endrule="quantile") # fast C code
b = runquantile(x, k, probs=0.4, endrule="func")     # slow R code
stopifnot(all(abs(a-b)<eps));

# test if moving windows forward and backward gives the same results
k=51;
a = runquantile(x     , k, probs=0.4)
b = runquantile(x[n:1], k, probs=0.4)
stopifnot(all(a[n:1]==b, na.rm=TRUE));

# test vector vs. matrix inputs, especially for the edge handling
nRow=200; k=25; nCol=10
x = rnorm(nRow,sd=30) + abs(seq(nRow)-n/4)
x[seq(1,nRow,10)] = NaN;              # add NANs
X = matrix(rep(x, nCol ), nRow, nCol) # replicate x in columns of X
```

```
  a = runquantile(x, k, probs=0.6)
  b = runquantile(X, k, probs=0.6)
  stopifnot(all(abs(a-b[,1])<eps));        # vector vs. 2D array
  stopifnot(all(abs(b[,1]-b[,nCol])<eps)); # compare rows within 2D array

  # Exhaustive testing of runquantile to standard R approach
  numeric.test = function (x, k) {
    probs=c(1, 25, 50, 75, 99)/100
    a = runquantile(x,k, c(0.3, 0.7), endrule="trim")
    b = t(apply(embed(x,k), 1, quantile, probs = c(0.3, 0.7), na.rm=TRUE))
    eps = .Machine$double.eps ^ 0.5
    stopifnot(all(abs(a-b)<eps));
  }
  n=50;
  x = rnorm(n,sd=30) + abs(seq(n)-n/4) # nice behaving data
  for(i in 2:5) numeric.test(x, i)     # test small window sizes
  for(i in 1:5) numeric.test(x, n-i+1) # test large window size
  x[seq(1,50,10)] = NaN;               # add NANs and repet the test
  for(i in 2:5) numeric.test(x, i)     # test small window sizes
  for(i in 1:5) numeric.test(x, n-i+1) # test large window size

  # Speed comparison
  ## Not run:
  x=runif(1e6); k=1e3+1;
  system.time(runquantile(x,k,0.5))    # Speed O(n*k)
  system.time(runmed(x,k))             # Speed O(n * log(k))

## End(Not run)
```

runsd *Standard Deviation of Moving Windows*

Description

Moving (aka running, rolling) Window's Standard Deviation calculated over a vector

Usage

```
runsd(x, k, center = runmean(x,k),
      endrule=c("sd", "NA", "trim", "keep", "constant", "func"),
      align = c("center", "left", "right"))
```

Arguments

x numeric vector of length n or matrix with n rows. If x is a matrix than each column will be processed separately.

k width of moving window; must be an integer between one and n. In case of even k's one will have to provide different center function, since runmed does not take even k's.

endrule — character string indicating how the values at the beginning and the end, of the data, should be treated. Only first and last k2 values at both ends are affected, where k2 is the half-bandwidth k2 = k %/% 2.

- "sd" - applies the sd function to smaller and smaller sections of the array. Equivalent to: for(i in 1:k2) out[i]=mad(x[1:(i+k2)]).
- "trim" - trim the ends; output array length is equal to length(x)-2*k2 (out = out[(k2+1):(n-k2)]). This option mimics output of apply (embed(x,k),1,FUN) and other related functions.
- "keep" - fill the ends with numbers from x vector (out[1:k2] = x[1:k2]). This option makes more sense in case of smoothing functions, kept here for consistency.
- "constant" - fill the ends with first and last calculated value in output array (out[1:k2] = out[k2+1])
- "NA" - fill the ends with NA's (out[1:k2] = NA)
- "func" - same as "mad" option except that implemented in R for testing purposes. Avoid since it can be very slow for large windows.

Similar to endrule in runmed function which has the following options: “c("median", "keep", "constant")” .

center — moving window center. Defaults to running mean (runmean function). Similar to center in mad function.

align — specifies whether result should be centered (default), left-aligned or right-aligned. If endrule="sd" then setting align to "left" or "right" will fall back on slower implementation equivalent to endrule="func".

Details

Apart from the end values, the result of y = runmad(x, k) is the same as “for(j=(1+k2):(n-k2)) y[j]=sd(x[(j-k2):(j+k2)], na.rm = TRUE)”. It can handle non-finite numbers like NaN's and Inf's (like mean(x, na.rm = TRUE)).

The main incentive to write this set of functions was relative slowness of majority of moving window functions available in R and its packages. With the exception of runmed, a running window median function, all functions listed in "see also" section are slower than very inefficient “apply(embed(x,k),1,FUN)” approach.

Value

Returns a numeric vector or matrix of the same size as x. Only in case of endrule="trim" the output vectors will be shorter and output matrices will have fewer rows.

Author(s)

Jarek Tuszynski (SAIC) <jaroslaw.w.tuszynski@saic.com>

See Also

Links related to:

- runsd - sd

- Other moving window functions from this package: runmin, runmax, runquantile, runmad and runmean
- generic running window functions: apply (embed(x,k), 1, FUN) (fastest), running from **gtools** package (extremely slow for this purpose), subsums from **magic** library can perform running window operations on data with any dimensions.

Examples

```
# show runmed function
k=25; n=200;
x = rnorm(n,sd=30) + abs(seq(n)-n/4)
col = c("black", "red", "green")
m=runmean(x, k)
y=runsd(x, k, center=m)
plot(x, col=col[1], main = "Moving Window Analysis Functions")
lines(m    , col=col[2])
lines(m-y/2, col=col[3])
lines(m+y/2, col=col[3])
lab = c("data", "runmean", "runmean-runsd/2", "runmean+runsd/2")
legend(0,0.9*n, lab, col=col, lty=1 )

# basic tests against apply/embed
eps = .Machine$double.eps ^ 0.5
k=25 # odd size window
a = runsd(x,k, endrule="trim")
b = apply(embed(x,k), 1, sd)
stopifnot(all(abs(a-b)<eps));
k=24 # even size window
a = runsd(x,k, endrule="trim")
b = apply(embed(x,k), 1, sd)
stopifnot(all(abs(a-b)<eps));

# test against loop approach
# this test works fine at the R prompt but fails during package check - need to investigate
k=25; n=200;
x = rnorm(n,sd=30) + abs(seq(n)-n/4) # create random data
x[seq(1,n,11)] = NaN;                # add NANs
k2 = k
k1 = k-k2-1
a = runsd(x, k)
b = array(0,n)
for(j in 1:n) {
  lo = max(1, j-k1)
  hi = min(n, j+k2)
  b[j] = sd(x[lo:hi], na.rm = TRUE)
}
#stopifnot(all(abs(a-b)<eps));

# compare calculation at array ends
k=25; n=100;
x = rnorm(n,sd=30) + abs(seq(n)-n/4)
a = runsd(x, k, endrule="sd" )   # fast C code
```

```
  b = runsd(x, k, endrule="func")  # slow R code
  stopifnot(all(abs(a-b)<eps));

  # test if moving windows forward and backward gives the same results
  k=51;
  a = runsd(x     , k)
  b = runsd(x[n:1], k)
  stopifnot(all(abs(a[n:1]-b)<eps));

  # test vector vs. matrix inputs, especially for the edge handling
  nRow=200; k=25; nCol=10
  x = rnorm(nRow,sd=30) + abs(seq(nRow)-n/4)
  x[seq(1,nRow,10)] = NaN;              # add NANs
  X = matrix(rep(x, nCol ), nRow, nCol) # replicate x in columns of X
  a = runsd(x, k)
  b = runsd(X, k)
  stopifnot(all(abs(a-b[,1])<eps));        # vector vs. 2D array
  stopifnot(all(abs(b[,1]-b[,nCol])<eps)); # compare rows within 2D array

  # speed comparison
  ## Not run:
  x=runif(1e5); k=51;                       # reduce vector and window sizes
  system.time(runsd( x,k,endrule="trim"))
  system.time(apply(embed(x,k), 1, sd))

## End(Not run)
```

---

sample.split — *Split Data into Test and Train Set*

Description

Split data from vector Y into two sets in predefined ratio while preserving relative ratios of different labels in Y. Used to split the data used during classification into train and test subsets.

Usage

```
sample.split( Y, SplitRatio = 2/3, group = NULL )
```

Arguments

Y — Vector of data labels. If there are only a few labels (as is expected) than relative ratio of data in both subsets will be the same.

SplitRatio — Splitting ratio:

- if (0<=SplitRatio<1) then SplitRatio fraction of points from Y will be set toTRUE
- if (SplitRatio==1) then one random point from Y will be set to TRUE
- if (SplitRatio>1) then SplitRatio number of points from Y will be set to TRUE

group	Optional vector/list used when multiple copies of each sample are present. In such a case group contains unique sample labels, marking all copies of the same sample with the same label, and the function tries to place all copies in either train or test subset. If provided than has to have the same length as Y.

Details

Function msc.sample.split is the old name of the sample.split function. To be retired soon. Note that the function differs from base::sample by first restricting the input data set to its unique values before generating the subset(s).

Value

Returns logical vector of the same length as Y with random SplitRatio*length(Y) elements set to TRUE.

Author(s)

Jarek Tuszynski (SAIC) <jaroslaw.w.tuszynski@saic.com>

See Also

- Similar to sample function.
- Variable group is used in the same way as f argument in split and INDEX argument in tapply

Examples

```
library(MASS)
data(cats)   # load cats data
Y = cats[,1] # extract labels from the data
msk = sample.split(Y, SplitRatio=3/4)
table(Y,msk)
t=sum( msk)  # number of elements in one class
f=sum(!msk)  # number of elements in the other class
stopifnot( round((t+f)*3/4) == t ) # test ratios

# example of using group variable
g = rep(seq(length(Y)/4), each=4); g[48]=12;
msk = sample.split(Y, SplitRatio=1/2, group=g)
table(Y,msk) # try to get correct split ratios ...
split(msk,g) # ... while keeping samples with the same group label together

# test results
print(paste( "All Labels numbers: total=",t+f,", train=",t,", test=",f,
      ", ratio=", t/(t+f) ) )
U = unique(Y)       # extract all unique labels
for( i in 1:length(U)) {  # check for all labels
  lab = (Y==U[i])   # mask elements that have label U[i]
  t=sum( msk[lab])  # number of elements with label U[i] in one class
  f=sum(!msk[lab])  # number of elements with label U[i] in the other class
  print(paste( "Label",U[i],"numbers: total=",t+f,", train=",t,", test=",f,
               ", ratio=", t/(t+f) ) )
}

# use results
train = cats[ msk,2:3]  # use output of sample.split to ...
test  = cats[!msk,2:3]  # create train and test subsets
z = lda(train, Y[msk])  # perform classification
table(predict(z, test)$class, Y[!msk]) # predicted & true labels

# see also LogitBoost example
```

`sumexact, cumsumexact` *Basic Sum Operations without Round-off Errors*

Description

Functions for performing basic sum operations without round-off errors

Usage

```
sumexact(..., na.rm = FALSE)
cumsumexact(x)
```

Arguments

`x`	numeric vector
`...`	numeric vector(s), numbers or other objects to be summed
`na.rm`	logical. Should missing values be removed?

Details

All three functions use full precision summation using multiple doubles for intermediate values. The sum of numbers x & y is a=x+y with error term b=error(a+b). That way a+b is equal exactly x+y, so sum of 2 numbers is stored as 2 or fewer values, which when added would under-flow. By extension sum of n numbers is calculated with intermediate results stored as array of numbers that can not be added without introducing an error. Only final result is converted to a single number

Value

Function `sumexact` returns single number. Function `cumsumexact` returns vector of the same length as `x`.

Author(s)

Jarek Tuszynski (SAIC) `<jaroslaw.w.tuszynski@saic.com>` based on code by Vadim Ogranovich, which is based on algorithms described in references, pointed out by Gabor Grothendieck.

References

Round-off error correction is based on: Shewchuk, Jonathan, *Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates*

McCullough, D.B., (1998) *Assessing the Reliability of Statistical Software, Part I*, The American Statistician, Vol. 52 No.

McCullough, D.B., (1999) *Assessing the Reliability of Statistical Software, Part II*, The American Statistician, Vol. 53 No 2

NIST Statistical Reference Datasets (StRD) website

See Also

- `sum` is faster but not error-save version of `sumexact`
- `cumsum` is equivalent to `cumsumexact`

Examples

```
x = c(1, 1e20, 1e40, -1e40, -1e20, -1)
a = sum(x);         print(a)
b = sumexact(x);    print(b)
stopifnot(b==0)
a = cumsum(x);      print(a)
b = cumsumexact(x); print(b)
stopifnot(b[6]==0)
```

`trapz` *Trapezoid Rule Numerical Integration*

Description

Computes the integral of Y with respect to X using trapezoid rule integration.

Usage

```
trapz(x, y)
```

Arguments

`x`	Sorted vector of x-axis values.
`y`	Vector of y-axis values.

Details

The function has only two lines:

```
idx = 2:length(x)
return (as.double( (x[idx] - x[idx-1]) %*% (y[idx] + y[idx-1])) / 2)
```

Value

Integral of Y with respect to X or area under the Y curve.

Note

Trapezoid rule is not the most accurate way of calculating integrals (it is exact for linear functions), for example Simpson's rule (exact for linear and quadratic functions) is more accurate.

Author(s)

Jarek Tuszynski (SAIC) <jaroslaw.w.tuszynski@saic.com>

References

D. Kincaid & W. Chaney (1991), *Numerical Analysis*, p.445

See Also

- integrate
- Matlab's trapz function (http://www.mathworks.com/access/helpdesk/help/techdoc/ref/trapz.html)

Examples

```
# integral of sine function in [0, pi] range suppose to be exactly 2.
# lets calculate it using 10 samples:
x = (1:10)*pi/10
trapz(x, sin(x))
# now lets calculate it using 1000 samples:
x = (1:1000)*pi/1000
trapz(x, sin(x))
```

Index