aroma.light

April 20, 2011

`1.  Calibration and Normalization`

1. Calibration and Normalization

Description

In this section we give *our* recommendation on how spotted two-color (or multi-color) microarray data is best calibrated and normalized.

Classical background subtraction

We do *not* recommend background subtraction in classical means where background is estimated by various image analysis methods. This means that we will only consider foreground signals in the analysis.

We estimate "background" by other means. In what is explain below, only a global background, that is, a global bias, is estimated and removed.

Multiscan calibration

In Bengtsson et al (2004) we give evidence that microarray scanners can introduce a significant bias in data. This bias, which is about 15-25 out of 65535, *will* introduce intensity dependency in the log-ratios, as explained in Bengtsson & Hössjer (2006).

In Bengtsson et al (2004) we find that this bias is stable across arrays (and a couple of months), but further research is needed in order to tell if this is true over a longer time period.

To calibrate signals for scanner biases, scan the same array at multiple PMT-settings at three or more (K >= 3) different PMT settings (preferably in decreasing order). While doing this, *do not adjust the laser power settings*. Also, do the multiscan *without* washing, cleaning or by other means changing the array between subsequent scans. Although not necessary, it is preferred that the array remains in the scanner between subsequent scans. This will simplify the image analysis since spot identification can be made once if images aligns perfectly.

After image analysis, read all K scans for the same array into the two matrices, one for the red and one for the green channel, where the K columns corresponds to scans and the N rows to the spots. It is enough to use foreground signals.

In order to multiscan calibrate the data, for each channel separately call `Xc <- calibrateMultiscan(X)` where `X` is the NxK matrix of signals for one channel across all scans. The calibrated signals are returned in the Nx1 matrix `Xc`.

Multiscan calibration may sometimes be skipped, especially if affine normalization is applied immediately after, but we do recommend that every lab check at least once if their scanner introduce bias. If the offsets in a scanner is already estimated from earlier multiscan analyses, or known by other means, they can readily be subtracted from the signals of each channel. If arrays are still multiscanned, it is possible to force the calibration method to fit the model with zero intercept (assuming the scanner offsets have been subtracted) by adding argument `center=FALSE`.

Affine normalization

In Bengtsson & Hössjer (2006), we carry out a detailed study on how biases in each channel introduce so called intensity-dependent log-ratios among other systematic artifacts. Data with (additive) bias in each channel is said to be *affinely* transformed. Data without such bias, is said to be *linearly* (proportionally) transform. Ideally, observed signals (data) is a linear (proportional) function of true gene expression levels.

We do *not* assume proportional observations. The scanner bias is real evidence that assuming linearity is not correct. Affine normalization corrects for affine transformation in data. Without control spots it is not possible to estimate the bias in each of the channels but only the relative bias such that after normalization the effective bias are the same in all channels. This is why we call it normalization and not calibration.

In its simplest form, affine normalization is done by `Xn <- normalizeAffine(X)` where `X` is a Nx2 matrix with the first column holds the foreground signals from the red channel and the second holds the signals from the green channel. If three- or four-channel data is used these are added the same way. The normalized data is returned as a Nx2 matrix `Xn`.

To normalize all arrays and all channels at once, one may put all data into one big NxK matrix where the K columns hold the all channels from the first array, then all channels from the second array and so on. Then `Xn <- normalizeAffine(X)` will return the across-array and across-channel normalized data in the NxK matrix `Xn` where the colunms are stored in the same order as in matrix `X`.

Equal effective bias in all channels is much better. First of all, any intensity-dependent bias in the log-ratios is removed *for all non-differentially expressed genes*. There is still an intensity-dependent bias in the log-ratios for differentially expressed genes, but this is now symmetric around log-ratio zero.

Affine normalization will (by default and recommended) normalize *all* arrays together and at once. This will guarantee that all arrays are "on the same scale". Thus, it *not* recommended to apply a classical between-array scale normalization afterward. Moreover, the average log-ratio will be zero after an affine normalization.

Note that an affine normalization will only remove curvature in the log-ratios at lower intensities. If a strong intensity-dependent bias at high intensities remains, this is most likely due to saturation effects, such as too high PMT settings or quenching.

Note that for a perfect affine normalization you *should* expect much higher noise levels in the *log-ratios* at lower intensities than at higher. It should also be approximately symmetric around zero log-ratio. In other words, *a strong fanning effect is a good sign*.

Due to different noise levels in red and green channels, different PMT settings in different channels, plus the fact that the minimum signal is zero, "odd shapes" may be seen in the log-ratio vs log-intensity graphs at lower intensities. Typically, these show themselves as non-symmetric in positive and negative log-ratios. Note that you should not see this at higher intensities.

If there is a strong intensity-dependent effect left after the affine normalization, we recommend, for now, that a subsequent curve-fit or quantile normalization is done. Which one, we do not know.

Why negative signals? By default, 5% of the normalized signals will have a non-positive signal in one or both channels. *This is on purpose*, although the exact number 5% is chosen by experience. The reason for introducing negative signals is that they are indeed expected. For instance, when measure a zero gene expression level, there is a chance that the observed value is (should be) negative due to measurement noise. (For this reason it is possible that the scanner manufacturers have introduced scanner bias on purpose to avoid negative signals, which then all would be truncated to zero.) To adjust the ratio (or number) of negative signals allowed, use for example `normalizeAffine(X, constraint=0.01)` for 1% negative signals. If set to zero (or `"max"`) only as much bias is removed such that no negative signals exist afterward. Note that this is also true if there were negative signals on beforehand.

Why not lowess normalization? Curve-fit normalization methods such as lowess normalization are basically designed based on linearity assumptions and will for this reason not correct for channel biases. Curve-fit normalization methods can by definition only be applied to one pair of channels at the time and do therefore require a subsequent between-array scale normalization, which is by the way very ad hoc.

Why not quantile normalization? Affine normalization can be though of a special case of quantile normalization that is more robust than the latter. See Bengtsson & Hössjer (2006) for details. Quantile normalization is probably better to apply than curve-fit normalization methods, but less robust than affine normalization, especially at extreme (low and high) intensities. For this reason, we do recommend to use affine normalization first, and if this is not satisfactory, quantile normalization may be applied.

Linear (proportional) normalization

If the channel offsets are zero, already corrected for, or estimated by other means, it is possible to normalize the data robustly by fitting the above affine model without intercept, that is, fitting a truly linear model. This is done adding argument `center=FALSE` when calling `normalizeAffine()`.

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

`aroma.light-package` *Package aroma.light*

Description

Methods for microarray analysis that take basic data types such as matrices and lists of vectors. These methods can be used standalone, be utilized in other packages, or be wrapped up in higher-level classes.

Requirements

This package requires the **R.oo** package [1].

Installation

To install this package, see `http://www.braju.com/R/`. Required packages are installed in the same way.

To get started

For scanner calibration:

1. see `calibrateMultiscan.matrix`() - scan the same array two or more times to calibrate for scanner effects and extended dynamical range.

To normalize multiple single-channel arrays all with the same number of probes/spots:

1. `normalizeAffine.matrix`() - normalizes, on the intensity scale, for differences in offset and scale between channels.
2. `normalizeQuantileRank.matrix`(), `normalizeQuantileSpline.matrix`() - normalizes, on the intensity scale, for differences in empirical distribution between channels.

To normalize multiple single-channel arrays with varying number probes/spots:

1. `normalizeQuantileRank.list`(), `normalizeQuantileSpline.list`() - normalizes, on the intensity scale, for differences in empirical distribution between channels.

To normalize two-channel arrays:

1. `normalizeAffine.matrix`() - normalizes, on the intensity scale, for differences in offset and scale between channels. This will also correct for intensity-dependent affects on the log scale.
2. `normalizeCurveFit.matrix`() - Classical intensity-dependent normalization, on the log scale, e.g. lowess normalization.

To normalize three or more channels:

1. `normalizeAffine.matrix`() - normalizes, on the intensity scale, for differences in offset and scale between channels. This will minimize the curvature on the log scale between any two channels.

Further readings

Several of the normalization methods proposed in [3]-[6] are available in this package.

How to cite this package

Whenever using this package, please cite [2] as

No citation information available.

Wishlist

Here is a list of features that would be useful, but which I have too little time to add myself. Contributions are appreciated.

- At the moment, nothing.

If you consider to contribute, make sure it is not already implemented by downloading the latest "devel" version!

License

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

References

Some of the reference below can be found at `http://www.maths.lth.se/bioinformatics/publications/`.

[1] H. Bengtsson, *The R.oo package - Object-Oriented Programming with References Using Standard R Code*, In Kurt Hornik, Friedrich Leisch and Achim Zeileis, editors, Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003), March 20-22, Vienna, Austria. `http://www.ci.tuwien.ac.at/Conferences/DSC-2003/Proceedings/`

[2] H. Bengtsson, *aroma - An R Object-oriented Microarray Analysis environment*, Preprints in Mathematical Sciences (manuscript in preparation), Mathematical Statistics, Centre for Mathematical Sciences, Lund University, 2004.

[3] Henrik Bengtsson and Ola Hössjer, *Methodological Study of Affine Transformations of Gene Expression Data*, Methodological study of affine transformations of gene expression data with proposed robust non-parametric multi-dimensional normalization method, BMC Bioinformatics, 2006, 7:100.

[4] H. Bengtsson, J. Vallon-Christersson and G. Jönsson, *Calibration and assessment of channel-specific biases in microarray data with extended dynamical range*, BMC Bioinformatics, 5:177, 2004.

[5] H. Bengtsson, R. Irizarry, B. Carvalho, and T. Speed, *Estimation and assessment of raw copy numbers at the single locus level*, Bioinformatics, 2008.

[6] H. Bengtsson, *Identification and normalization of plate effects in cDNA microarray data*, Preprints in Mathematical Sciences, 2002:28, Mathematical Statistics, Centre for Mathematical Sciences, Lund University, 2002.

`averageQuantile.list`

Gets the average empirical distribution

Description

Gets the average empirical distribution for a set of samples of different sizes.

Usage

```
## S3 method for class 'list':
averageQuantile(X, ...)
```

Arguments

`X`	a `list` with `numeric` `vector`s. The `vector`s may be of different lengths.
`...`	Not used.

Value

Returns a `numeric` `vector` of length equal to the longest `vector` in argument `X`.

Missing values

Missing values are excluded.

Author(s)

Parts adopted from Gordon Smyth (`http://www.statsci.org/`) in 2002 \& 2006. Original code by Ben Bolstad at Statistics Department, University of California.

See Also

`*normalizeQuantileRank()`. `*normalizeQuantileSpline()`. `quantile`.

`backtransformAffine.matrix`
Reverse affine transformation

Description

Reverse affine transformation.

Usage

```
## S3 method for class 'matrix':
backtransformAffine(X, a=NULL, b=NULL, project=FALSE, ...)
```

Arguments

`X`	An NxK `matrix` containing data to be backtransformed.
`a`	A scalar, `vector`, a `matrix`, or a `list`. First, if a `list`, it is assumed to contained the elements `a` and `b`, which are the used as if they were passed as seperate arguments. If a `vector`, a matrix of size NxK is created which is then filled *row by row* with the values in the vector. Commonly, the vector is of length K, which means that the matrix will consist of copies of this vector stacked on top of each other. If a `matrix`, a matrix of size NxK is created which is then filled *column by column* with the values in the matrix (collected column by column. Commonly, the matrix is of size NxK, or NxL with L < K and then the resulting matrix consists of copies sitting next to each other. The resulting NxK matrix is subtracted from the NxK matrix `X`.

`b`	A scalar, `vector`, a `matrix`. A NxK matrix is created from this argument. For details see argument `a`. The NxK matrix `X-a` is divided by the resulting NxK matrix.
`project`	returned (K values per data point are returned). If `TRUE`, the backtransformed values "`(X-a)/b`" are projected onto the line L(a,b) so that all columns will be identical.
`...`	Not used.

Value

The "`(X-a)/b`" backtransformed NxK `matrix` is returned. If `project` is `TRUE`, an Nx1 `matrix` is returned, because all columns are identical anyway.

Missing values

Missing values remain missing values. If projected, data points that contain missing values are projected without these.

See Also

For more information see `matrix`.

Examples

```
X <- matrix(1:8, nrow=4, ncol=2)
X[2,2] <- NA

print(X)

# Returns a 4x2 matrix
print(backtransformAffine(X, a=c(1,5)))

# Returns a 4x2 matrix
print(backtransformAffine(X, b=c(1,1/2)))

# Returns a 4x2 matrix
print(backtransformAffine(X, a=matrix(1:4,ncol=1)))

# Returns a 4x2 matrix
print(backtransformAffine(X, a=matrix(1:3,ncol=1)))

# Returns a 4x2 matrix
print(backtransformAffine(X, a=matrix(1:2,ncol=1), b=c(1,2)))

# Returns a 4x1 matrix
print(backtransformAffine(X, b=c(1,1/2), project=TRUE))

# If the columns of X are identical, and a identity
# backtransformation is applied and projected, the
# same matrix is returned.
X <- matrix(1:4, nrow=4, ncol=3)
Y <- backtransformAffine(X, b=c(1,1,1), project=TRUE)
print(X)
print(Y)
stopifnot(sum(X[,1]-Y) <= .Machine$double.eps)
```

```
# If the columns of X are identical, and a identity
# backtransformation is applied and projected, the
# same matrix is returned.
X <- matrix(1:4, nrow=4, ncol=3)
X[,2] <- X[,2]*2; X[,3] <- X[,3]*3;
print(X)
Y <- backtransformAffine(X, b=c(1,2,3))
print(Y)
Y <- backtransformAffine(X, b=c(1,2,3), project=TRUE)
print(Y)
stopifnot(sum(X[,1]-Y) <= .Machine$double.eps)
```

`backtransformPrincipalCurve.matrix`

Reverse transformation of principal-curve fit

Description

Reverse transformation of principal-curve fit.

Usage

```
## S3 method for class 'matrix':
backtransformPrincipalCurve(X, fit, dimensions=NULL, targetDimension=NULL, ...)
```

Arguments

`X`	An NxK `matrix` containing data to be backtransformed.
`fit`	An MxL principal-curve fit object of class `principal.curve` as returned by `*fitPrincipalCurve()`. Typically $L = K$, but not always.
`dimensions`	An (optional) subset of of D dimensions all in [1,L] to be returned (and backtransform).
`targetDimension`	An (optional) index specifying the dimension in [1,L] to be used as the target dimension of the `fit`. More details below.
`...`	Passed internally to `smooth.spline`.

Value

The backtransformed NxK (or NxD) `matrix`.

Target dimension

By default, the backtransform is such that afterward the signals are approximately proportional to the (first) principal curve as fitted by `*fitPrincipalCurve()`. This scale and origin of this principal curve is not uniquely defined. If `targetDimension` is specified, then the backtransformed signals are approximately proportional to the signals of the target dimension, and the signals in the target dimension are unchanged.

Subsetting dimensions

Argument `dimensions` can be used to backtransform a subset of dimensions (K) based on a subset of the fitted dimensions (L). If $K = L$, then both `X` and `fit` is subsetted. If $K <> L$, then it is assumed that `X` is already subsetted/expanded and only `fit` is subsetted.

See Also

*`fitPrincipalCurve()`

Examples

```
# Consider the case where K=4 measurements have been done
# for the same underlying signals 'x'.  The different measurements
# have different systematic variation
#
#   y_k = f(x_k) + eps_k; k = 1,...,K.
#
# In this example, we assume non-linear measurement functions
#
#   f(x) = a + b*x + x^c + eps(b*x)
#
# where 'a' is an offset, 'b' a scale factor, and 'c' an exponential.
# We also assume heteroscedastic zero-mean noise with standard
# deviation proportional to the rescaled underlying signal 'x'.
#
# Furthermore, we assume that measurements k=2 and k=3 undergo the
# same transformation, which may illustrate that the come from
# the same batch. However, when *fitting* the model below we
# will assume they are independent.

# Transforms
a <- c(2, 15, 15,   3)
b <- c(2,  3,  3,   4)
c <- c(1,  2,  2, 1/2)
K <- length(a)

# The true signal
N <- 1000
x <- rexp(N)

# The noise
bX <- outer(b,x)
E <- apply(bX, MARGIN=2, FUN=function(x) rnorm(K, mean=0, sd=0.1*x))

# The transformed signals with noise
Xc <- t(sapply(c, FUN=function(c) x^c))
Y <- a + bX + Xc + E
Y <- t(Y)



# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Fit principal curve
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Fit principal curve through Y = (y_1, y_2, ..., y_K)
```

```
fit <- fitPrincipalCurve(Y)

# Flip direction of 'lambda'?
rho <- cor(fit$lambda, Y[,1], use="complete.obs")
flip <- (rho < 0)
if (flip) {
  fit$lambda <- max(fit$lambda, na.rm=TRUE)-fit$lambda
}

L <- ncol(fit$s)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Backtransform data according to model fit
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Backtransform toward the principal curve (the "common scale")
YN1 <- backtransformPrincipalCurve(Y, fit=fit)
stopifnot(ncol(YN1) == K)


# Backtransform toward the first dimension
YN2 <- backtransformPrincipalCurve(Y, fit=fit, targetDimension=1)
stopifnot(ncol(YN2) == K)


# Backtransform toward the last (fitted) dimension
YN3 <- backtransformPrincipalCurve(Y, fit=fit, targetDimension=L)
stopifnot(ncol(YN3) == K)


# Backtransform toward the third dimension (dimension by dimension)
# Note, this assumes that K == L.
YN4 <- Y
for (cc in 1:L) {
  YN4[,cc] <- backtransformPrincipalCurve(Y, fit=fit,
                                  targetDimension=1, dimensions=cc)
}
stopifnot(identical(YN4, YN2))


# Backtransform a subset toward the first dimension
# Note, this assumes that K == L.
YN5 <- backtransformPrincipalCurve(Y, fit=fit,
                               targetDimension=1, dimensions=2:3)
stopifnot(identical(YN5, YN2[,2:3]))
stopifnot(ncol(YN5) == 2)


# Extract signals from measurement #2 and backtransform according
# its model fit.  Signals are standardized to target dimension 1.
y6 <- Y[,2,drop=FALSE]
yN6 <- backtransformPrincipalCurve(y6, fit=fit, dimensions=2,
                                               targetDimension=1)
stopifnot(identical(yN6, YN2[,2,drop=FALSE]))
stopifnot(ncol(yN6) == 1)


# Extract signals from measurement #2 and backtransform according
# the the model fit of measurement #3 (because we believe these
# two have undergone very similar transformations.
# Signals are standardized to target dimension 1.
y7 <- Y[,2,drop=FALSE]
yN7 <- backtransformPrincipalCurve(y7, fit=fit, dimensions=3,
                                              targetDimension=1)
stopifnot(ncol(yN7) == 1)
stopifnot(cor(yN7, yN6) > 0.9999)
```

`calibrateMultiscan.matrix` *Weighted affine calibration of a multiple re-scanned channel*

Description

Weighted affine calibration of a multiple re-scanned channel.

Usage

```
## S3 method for class 'matrix':
calibrateMultiscan(X, weights=NULL, typeOfWeights=c("datapoint"), method="L1", c
```

Arguments

`X`	An NxK `matrix` (K>=2) where the columns represent the multiple scans of one channel (a two-color array contains two channels) to be calibrated.
`weights`	If `NULL`, non-weighted normalization is done. If data-point weights are used, this should be a `vector` of length N of data point weights used when estimating the normalization function.
`typeOfWeights`	A `character` string specifying the type of weights given in argument `weights`.
`method`	A `character` string specifying how the estimates are robustified. See `*iwpca()` for all accepted values.
`constraint`	Constraint making the bias parameters identifiable. See `*fitIWPCA()` for more details.
`satSignal`	Signals equal to or above this threshold is considered saturated signals.
`...`	Other arguments passed to `*fitIWPCA()` and in turn `*iwpca()`, e.g. `center` (see below).
`average`	A `function` to calculate the average signals between calibrated scans.
`deviance`	A `function` to calculate the deviance of the signals between calibrated scans.
`project`	If `TRUE`, the calibrated data points projected onto the diagonal line, otherwise not. Moreover, if `TRUE`, argument `average` is ignored.
`.fitOnly`	If `TRUE`, the data will not be back-transform.

Details

Fitting is done by iterated re-weighted principal component analysis (IWPCA).

Value

If `average` is specified or `project` is `TRUE`, an Nx1 `matrix` is returned, otherwise an NxK `matrix` is returned. If `deviance` is specified, a deviance Nx1 `matrix` is returned as attribute `deviance`. In addition, the fitted model is returned as attribute `modelFit`.

Negative, non-positive, and saturated values

Affine multiscan calibration applies also to negative values, which are therefor also calibrated, if they exist.

Saturated signals in any scan are set to `NA`. Thus, they will not be used to estimate the calibration function, nor will they affect an optional projection.

Missing values

Only observations (rows) in `X` that contain all finite values are used in the estimation of the alibration functions. Thus, observations can be excluded by setting them to `NA`.

Weighted normalization

Each data point/observation, that is, each row in `X`, which is a vector of length K, can be assigned a weight in [0,1] specifying how much it should *affect the fitting of the calibration function*. Weights are given by argument `weights`, which should be a `numeric` `vector` of length N. Regardless of weights, all data points are *calibrated* based on the fitted calibration function.

Robustness

By default, the model fit of multiscan calibration is done in L_1 (`method="L1"`). This way, outliers affect the parameter estimates less than ordinary least-square methods.

When calculating the average calibrated signal from multiple scans, by default the median is used, which further robustify against outliers.

For further robustness, downweight outliers such as saturated signals, if possible.

Tukey's biweight function is supported, but not used by default because then a "bandwidth" parameter has to selected. This can indeed be done automatically by estimating the standard deviation, for instance using MAD. However, since scanner signals have heteroscedastic noise (standard deviation is approximately proportional to the non-logged signal), Tukey's bandwidth parameter has to be a function of the signal too, cf. `loess`. We have experimented with this too, but found that it does not significantly improve the robustness compared to L_1. Moreover, using Tukey's biweight as is, that is, assuming homoscedastic noise, seems to introduce a (scale dependent) bias in the estimates of the offset terms.

Using a known/previously estimated offset

If the scanner offsets can be assumed to be known, for instance, from prior multiscan analyses on the scanner, then it is possible to fit the scanner model with no (zero) offset by specifying argument `center=FALSE`. Note that you cannot specify the offset. Instead, subtract it from all signals before calibrating, e.g. `Xc <- calibrateMultiscan(X-e, center=FALSE)` where `e` is the scanner offset (a scalar). You can assert that the model is fitted without offset by `stopifnot(all(attr(Xc, "modelFit")$adiag == 0))`.

Author(s)

Henrik Bengtsson (http://www.braju.com/R/)

References

[1] H. Bengtsson, J. Vallon-Christersson and G. Jönsson, *Calibration and assessment of channel-specific biases in microarray data with extended dynamical range*, BMC Bioinformatics, 5:177, 2004.

See Also

`1. Calibration and Normalization`. `*normalizeAffine()`. For more information see `matrix`.

Examples

```
## Not run: # For an example, see help(normalizeAffine).
```

`callNaiveGenotypes.numeric`

Calls genotypes in a normal sample

Description

Calls genotypes in a normal sample.

Usage

```
## S3 method for class 'numeric':
callNaiveGenotypes(y, cn=rep(2, length(y)), flavor=c("density"), ..., modelFit=N
```

Arguments

`y`	A `numeric vector` of length J containing allele B fractions for a normal sample.
`cn`	An optional `numeric vector` of length J specifying the true total copy number in $\{0, 1, 2, NA\}$ at each locus. This can be used to specify which loci are diploid and which are not, e.g. autosomal and sex chromosome copy numbers.
`flavor`	A `character` string specifying the type of algorithm used.
`...`	Additional arguments passed to `*fitNaiveGenotypes()`.
`modelFit`	A optional model fit as returned by `*fitNaiveGenotypes()`.
`verbose`	A `logical` or a `Verbose` object.

Value

Returns a `numeric vector` of length J containing the genotype calls in allele B fraction space, that is, in [0,1] where 1/2 corresponds to a heterozygous call, and 0 and 1 corresponds to homozygous A and B, respectively. Non called genotypes have value `NA`.

Missing and non-finite values

A missing value always gives a missing (`NA`) genotype call. Negative infinity (-`Inf`) always gives genotype call 0. Positive infinity (+`Inf`) always gives genotype call 1.

Author(s)

Henrik Bengtsson (http://www.braju.com/R/)

See Also

Internally `*fitNaiveGenotypes()` is used to identify the thresholds.

Examples

```
layout(matrix(1:3, ncol=1))
par(mar=c(2,4,4,1)+0.1)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# A bimodal distribution
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
xAA <- rnorm(n=10000, mean=0, sd=0.1)
xBB <- rnorm(n=10000, mean=1, sd=0.1)
x <- c(xAA,xBB)
fit <- findPeaksAndValleys(x)
print(fit)
calls <- callNaiveGenotypes(x, cn=rep(1,length(x)), verbose=-20)
xc <- split(x, calls)
print(table(calls))
xx <- c(list(x),xc)
plotDensity(xx, adjust=1.5, lwd=2, col=seq(along=xx), main="(AA,BB)")
abline(v=fit$x)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# A trimodal distribution with missing values
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
xAB <- rnorm(n=10000, mean=1/2, sd=0.1)
x <- c(xAA,xAB,xBB)
x[sample(length(x), size=0.05*length(x))] <- NA;
x[sample(length(x), size=0.01*length(x))] <- -Inf;
x[sample(length(x), size=0.01*length(x))] <- +Inf;
fit <- findPeaksAndValleys(x)
print(fit)
calls <- callNaiveGenotypes(x)
xc <- split(x, calls)
print(table(calls))
xx <- c(list(x),xc)
plotDensity(xx, adjust=1.5, lwd=2, col=seq(along=xx), main="(AA,AB,BB)")
abline(v=fit$x)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# A trimodal distribution with clear separation
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
xAA <- rnorm(n=10000, mean=0, sd=0.02)
xAB <- rnorm(n=10000, mean=1/2, sd=0.02)
xBB <- rnorm(n=10000, mean=1, sd=0.02)
x <- c(xAA,xAB,xBB)
fit <- findPeaksAndValleys(x)
print(fit)
calls <- callNaiveGenotypes(x)
xc <- split(x, calls)
print(table(calls))
```

```
xx <- c(list(x),xc)
plotDensity(xx, adjust=1.5, lwd=2, col=seq(along=xx), main="(AA',AB',BB')")
abline(v=fit$x)
```

---

distanceBetweenLines

Finds the shortest distance between two lines

Description

Finds the shortest distance between two lines.

Consider the two lines

$x(s) = a_x + b_x * s$ and $y(t) = a_y + b_y * t$

in an K-space where the offset and direction vectors are a_x and b_x (in R^K) that define the line $x(s)$ (s is a scalar). Similar for the line $y(t)$. This function finds the point (s, t) for which $|x(s) - x(t)|$ is minimal.

Usage

```
## Default S3 method:
distanceBetweenLines(ax, bx, ay, by, ...)
```

Arguments

ax,bx	Offset and direction vector of length K for line z_x.
ay,by	Offset and direction vector of length K for line z_y.
...	Not used.

Value

Returns the a list containing

ax,bx	The given line $x(s)$.
ay,by	The given line $y(t)$.
s,t	The values of s and t such that $|x(s) - y(t)|$ is minimal.
xs,yt	The values of $x(s)$ and $y(t)$ at the optimal point (s, t).
distance	The distance between the lines, i.e. $|x(s) - y(t)|$ at the optimal point (s, t).

Author(s)

Henrik Bengtsson (http://www.braju.com/R/)

References

[1] M. Bard and D. Himel, *The Minimum Distance Between Two Lines in n-Space*, September 2001, Advisor Dennis Merino.
[2] Dan Sunday, *Distance between Lines and Segments with their Closest Point of Approach*, http://geometryalgorithms.com/Archive/algorithm_0106/.

Examples

```
for (zzz in 0) {

# This example requires plot3d() in R.basic [http://www.braju.com/R/]
if (!require(R.basic)) break

layout(matrix(1:4, nrow=2, ncol=2, byrow=TRUE))

############################################################
# Lines in two-dimensions
############################################################
x <- list(a=c(1,0), b=c(1,2))
y <- list(a=c(0,2), b=c(1,1))
fit <- distanceBetweenLines(ax=x$a, bx=x$b, ay=y$a, by=y$b)

xlim <- ylim <- c(-1,8)
plot(NA, xlab="", ylab="", xlim=ylim, ylim=ylim)

# Highlight the offset coordinates for both lines
points(t(x$a), pch="+", col="red")
text(t(x$a), label=expression(a[x]), adj=c(-1,0.5))
points(t(y$a), pch="+", col="blue")
text(t(y$a), label=expression(a[y]), adj=c(-1,0.5))

v <- c(-1,1)*10;
xv <- list(x=x$a[1]+x$b[1]*v, y=x$a[2]+x$b[2]*v)
yv <- list(x=y$a[1]+y$b[1]*v, y=y$a[2]+y$b[2]*v)

lines(xv, col="red")
lines(yv, col="blue")

points(t(fit$xs), cex=2.0, col="red")
text(t(fit$xs), label=expression(x(s)), adj=c(+2,0.5))
points(t(fit$yt), cex=1.5, col="blue")
text(t(fit$yt), label=expression(y(t)), adj=c(-1,0.5))
print(fit)


############################################################
# Lines in three-dimensions
############################################################
x <- list(a=c(0,0,0), b=c(1,1,1))  # The 'diagonal'
y <- list(a=c(2,1,2), b=c(2,1,3))  # A 'fitted' line
fit <- distanceBetweenLines(ax=x$a, bx=x$b, ay=y$a, by=y$b)

xlim <- ylim <- zlim <- c(-1,3)
dummy <- t(c(1,1,1))*100;

# Coordinates for the lines in 3d
v <- seq(-10,10, by=1);
xv <- list(x=x$a[1]+x$b[1]*v, y=x$a[2]+x$b[2]*v, z=x$a[3]+x$b[3]*v)
yv <- list(x=y$a[1]+y$b[1]*v, y=y$a[2]+y$b[2]*v, z=y$a[3]+y$b[3]*v)

for (theta in seq(30,140,length=3)) {
  plot3d(dummy, theta=theta, phi=30, xlab="", ylab="", zlab="",
                             xlim=ylim, ylim=ylim, zlim=zlim)
```

```
    # Highlight the offset coordinates for both lines
    points3d(t(x$a), pch="+", col="red")
    text3d(t(x$a), label=expression(a[x]), adj=c(-1,0.5))
    points3d(t(y$a), pch="+", col="blue")
    text3d(t(y$a), label=expression(a[y]), adj=c(-1,0.5))

    # Draw the lines
    lines3d(xv, col="red")
    lines3d(yv, col="blue")

    # Draw the two points that are closest to each other
    points3d(t(fit$xs), cex=2.0, col="red")
    text3d(t(fit$xs), label=expression(x(s)), adj=c(+2,0.5))
    points3d(t(fit$yt), cex=1.5, col="blue")
    text3d(t(fit$yt), label=expression(y(t)), adj=c(-1,0.5))

    # Draw the distance between the two points
    lines3d(rbind(fit$xs,fit$yt), col="purple", lwd=2)
  }

  print(fit)

  } # for (zzz in 0)
  rm(zzz)
```

`fitIWPCA.matrix` *Robust fit of linear subspace through multidimensional data*

Description

Robust fit of linear subspace through multidimensional data.

Usage

```
## S3 method for class 'matrix':
fitIWPCA(X, constraint=c("diagonal", "baseline", "max"), baselineChannel=NULL, .
```

Arguments

`X`	NxK `matrix` where N is the number of observations and K is the number of dimensions (channels).
`constraint`	A `character` string or a `numeric` value. If `character` it specifies which additional contraint to be used to specify the offset parameters along the fitted line; If `"diagonal"`, the offset vector will be a point on the line that is closest to the diagonal line (1,...,1). With this constraint, all bias parameters are identifiable. If `"baseline"` (requires argument `baselineChannel`), the estimates are such that of the bias and scale parameters of the baseline channel is 0 and 1, respectively. With this constraint, all bias parameters are identifiable. If `"max"`, the offset vector will the point on the line that is as "great" as possible, but still such that each of its components is less than the corresponding minimal signal. This will guarantee that no negative signals are created

	in the backward transformation. If `numeric` value, the offset vector will the point on the line such that after applying the backward transformation there are `constraint*N`. Note that `constraint==0` corresponds approximately to `constraint=="max"`. With the latter two constraints, the bias parameters are only identifiable modulo the fitted line.
`baselineChannel`	Index of channel toward which all other channels are conform. This argument is required if `constraint=="baseline"`. This argument is optional if `constraint=="diagonal"` and then the scale factor of the baseline channel will be one. The estimate of the bias parameters is not affected in this case. Defaults to one, if missing.
`...`	Additional arguments accepted by `*iwpca()`. For instance, a N `vector` of weights for each observation may be given, otherwise they get the same weight.
`aShift, Xmin`	For internal use only.

Details

This method uses re-weighted principal component analysis (IWPCA) to fit a the nodel $y_n = a + bx_n + eps_n$ where y_n, a, b, and eps_n are vector of the K and x_n is a scalar.

The algorithm is: For iteration i: 1) Fit a line L through the data close using weighted PCA with weights $\{w_n\}$. Let $r_n = \{r_{n,1}, ..., r_{n,K}\}$ be the K principal components. 2) Update the weights as $w_n < -1/\sum_2^K (r_{n,k} + \epsilon_r)$ where we have used the residuals of all but the first principal component. 3) Find the point a on L that is closest to the line $D = (1, 1, ..., 1)$. Similarily, denote the point on D that is closest to L by $t = a * (1, 1, ..., 1)$.

Value

Returns a `list` that contains estimated parameters and algorithm details;

`a`	A `double vector` $(a[1], ..., a[K])$with offset parameter estimates. It is made identifiable according to argument `constraint`.
`b`	A `double vector` $(b[1], ..., b[K])$with scale parameter estimates. It is made identifiable by constraining `b[baselineChannel] == 1`. These estimates are idependent of argument `constraint`.
`adiag`	If identifiability constraint `"diagonal"`, a `double vector` $(adiag[1], ..., adiag[K])$, where $adiag[1] = adiag[2] = ...adiag[K]$, specifying the point on the diagonal line that is closest to the fitted line, otherwise the zero vector.
`eigen`	A KxK `matrix` with columns of eigenvectors.
`converged`	`TRUE` if the algorithm converged, otherwise `FALSE`.
`nbrOfIterations`	The number of iterations for the algorithm to converge, or zero if it did not converge.
`t0`	Internal parameter estimates, which contains no more information than the above listed elements.
`t`	Always `NULL`.

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

See Also

This is an internal method used by the `*calibrateMultiscan()` and `*normalizeAffine()` methods. Internally the function `*iwpca()` is used to fit a line through the data cloud and the function `distanceBetweenLines`() to find the closest point to the diagonal (1,1,...,1).

`fitNaiveGenotypes.numeric` *Fit naive genotype model from a normal sample*

Description

Fit naive genotype model from a normal sample.

Usage

```
## S3 method for class 'numeric':
fitNaiveGenotypes(y, cn=rep(2, length(y)), subsetToFit=NULL, flavor=c("density")
```

Arguments

`y`	A `numeric vector` of length J containing allele B fractions for a normal sample.
`cn`	An optional `numeric vector` of length J specifying the true total copy number in $\{0, 1, 2, NA\}$ at each locus. This can be used to specify which loci are diploid and which are not, e.g. autosomal and sex chromosome copy numbers.
`subsetToFit`	An optional `integer` or `logical vector` specifying which loci should be used for estimating the model. If `NULL`, all loci are used.
`flavor`	A `character` string specifying the type of algorithm used.
`adjust`	A postive `double` specifying the amount smoothing for the empirical density estimator.
`...`	Additional arguments passed to `findPeaksAndValleys`().
`censorAt`	A `double vector` of length two specifying the range for which values are considered finite. Values below (above) this range are treated as -`Inf` (+`Inf`).
`verbose`	A `logical` or a `Verbose` object.

Value

Returns a `list` of `list`s.

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

See Also

To call genotypes see `*callNaiveGenotypes()`. Internally `findPeaksAndValleys`() is used to identify the thresholds.

fitPrincipalCurve.matrix

Fit a principal curve in K dimensions

Description

Fit a principal curve in K dimensions.

Usage

```
## S3 method for class 'matrix':
fitPrincipalCurve(X, ..., verbose=FALSE)
```

Arguments

`X`	An NxK `matrix` (K>=2) where the columns represent the dimension.
`...`	Other arguments passed to `principal.curve`.
`verbose`	A `logical` or a `Verbose` object.

Value

Returns a principal.curve object (which is a `list`). See `principal.curve` for more details.

Missing values

The estimation of the affine normalization function will only be made based on complete observations, i.e. observations that contains no `NA` values in any of the channels.

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

References

[1] Hastie, T. and Stuetzle, W, *Principal Curves*, JASA, 1989.

See Also

`*backtransformPrincipalCurve().` `principal.curve`.

Examples

```
# Simulate data from the model y <- a + bx + x^c + eps(bx)
J <- 1000
x <- rexp(J)
a <- c(2,15,3)
b <- c(2,3,4)
c <- c(1,2,1/2)
bx <- outer(b,x)
xc <- t(sapply(c, FUN=function(c) x^c))
eps <- apply(bx, MARGIN=2, FUN=function(x) rnorm(length(b), mean=0, sd=0.1*x))
```

```
y <- a + bx + xc + eps
y <- t(y)

# Fit principal curve through (y_1, y_2, y_3)
fit <- fitPrincipalCurve(y, verbose=TRUE)

# Flip direction of 'lambda'?
rho <- cor(fit$lambda, y[,1], use="complete.obs")
flip <- (rho < 0)
if (flip) {
  fit$lambda <- max(fit$lambda, na.rm=TRUE)-fit$lambda
}


# Backtransform (y_1, y_2, y_3) to be proportional to each other
yN <- backtransformPrincipalCurve(y, fit=fit)

# Same backtransformation dimension by dimension
yN2 <- y
for (cc in 1:ncol(y)) {
  yN2[,cc] <- backtransformPrincipalCurve(y, fit=fit, dimensions=cc)
}
stopifnot(identical(yN2, yN))


xlim <- c(0, 1.04*max(x))
ylim <- range(c(y,yN), na.rm=TRUE)


# Display raw and backtransform data
layout(matrix(1:4, nrow=2, byrow=TRUE))
par(mar=c(4,4,2,1)+0.1)
for (rr in 1:2) {
  ylab <- substitute(y[c], env=list(c=rr))
  for (cc in 2:3) {
    if (cc == rr) {
      plot.new()
      next
    }
    xlab <- substitute(y[c], env=list(c=cc))
    plot(NA, xlim=ylim, ylim=ylim, xlab=xlab, ylab=ylab)
    abline(a=0, b=1, lty=2)
    points(y[,c(cc,rr)])
    points(yN[,c(cc,rr)], col="tomato")
  }
}


layout(matrix(1:4, nrow=2, byrow=TRUE))
par(mar=c(4,4,2,1)+0.1)
for (cc in 1:3) {
  ylab <- substitute(y[c], env=list(c=cc))
  plot(NA, xlim=xlim, ylim=ylim, xlab="x", ylab=ylab)
  points(x, y[,cc])
  points(x, yN[,cc], col="tomato")
}
```

fitXYCurve.matrix *Fitting a smooth curve through paired (x,y) data*

Description

Fitting a smooth curve through paired (x,y) data.

Usage

```
## S3 method for class 'matrix':
fitXYCurve(X, weights=NULL, typeOfWeights=c("datapoint"), method=c("loess", "low
```

Arguments

X	An Nx2 matrix where the columns represent the two channels to be normalized.
weights	If NULL, non-weighted normalization is done. If data-point weights are used, this should be a vector of length N of data point weights used when estimating the normalization function.
typeOfWeights	A character string specifying the type of weights given in argument weights.
method	character string specifying which method to use when fitting the intensity-dependent function. Supported methods: "loess" (better than lowess), "lowess" (classic; supports only zero-one weights), "spline" (more robust than lowess at lower and upper intensities; supports only zero-one weights), "robustSpline" (better than spline).
bandwidth	A double value specifying the bandwidth of the estimator used.
satSignal	Signals equal to or above this threshold will not be used in the fitting.
...	Not used.

Value

A named list structure of class XYCurve.

Missing values

The estimation of the function will only be made based on complete non-saturated observations, i.e. observations that contains no NA values nor saturated values as defined by satSignal.

Weighted normalization

Each data point, that is, each row in X, which is a vector of length 2, can be assigned a weight in [0,1] specifying how much it should *affect the fitting of the affine normalization function*. Weights are given by argument weights, which should be a numeric vector of length N.

Note that the lowess and the spline method only support zero-one {0,1} weights. For such methods, all weights that are less than a half are set to zero.

Details on loess

For loess, the arguments family="symmetric", degree=1, span=3/4, control=loess.control(trac iterations=5, surface="direct") are used.

Author(s)

Henrik Bengtsson (http://www.braju.com/R/)

Examples

```
 # Simulate data from the model y <- a + bx + x^c + eps(bx)
x <- rexp(1000)
a <- c(2,15)
b <- c(2,1)
c <- c(1,2)
bx <- outer(b,x)
xc <- t(sapply(c, FUN=function(c) x^c))
eps <- apply(bx, MARGIN=2, FUN=function(x) rnorm(length(x), mean=0, sd=0.1*x))
Y <- a + bx + xc + eps
Y <- t(Y)

lim <- c(0,70)
plot(Y, xlim=lim, ylim=lim)

# Fit principal curve through a subset of (y_1, y_2)
subset <- sample(nrow(Y), size=0.3*nrow(Y))
fit <- fitXYCurve(Y[subset,], bandwidth=0.2)

lines(fit, col="red", lwd=2)

# Backtransform (y_1, y_2) keeping y_1 unchanged
YN <- backtransformXYCurve(Y, fit=fit)
points(YN, col="blue")
abline(a=0, b=1, col="red", lwd=2)
```

`iwpca.matrix` *Fits an R-dimensional hyperplane using iterative re-weighted PCA*

Description

Fits an R-dimensional hyperplane using iterative re-weighted PCA.

Usage

```
## S3 method for class 'matrix':
iwpca(X, w=NULL, R=1, method=c("symmetric", "bisquare", "tricube", "L1"), maxIte
```

Arguments

X	N-times-K matrix where N is the number of observations and K is the number of dimensions.
w	An N vector of weights for each row (observation) in the data matrix. If NULL, all observations get the same weight.
R	Number of principal components to fit. By default a line is fitted.

`method`	If `"symmetric"` (or `"bisquare"`), Tukey's biweight is used. If `"tricube"`, the tricube weight is used. If `"L1"`, the model is fitted in L_1. If a `function`, it is used to calculate weights for next iteration based on the current iteration's residuals.
`maxIter`	Maximum number of iterations.
`acc`	The (Euclidean) distance between two subsequent parameters fit for which the algorithm is considered to have converged.
`reps`	Small value to be added to the residuals before the the weights are calculated based on their inverse. This is to avoid infinite weights.
`fit0`	A `list` containing elements `vt` and `pc` specifying an initial fit. If `NULL`, the initial guess will be equal to the (weighted) PCA fit.
`...`	Additional arguments accepted by `*wpca()`.

Details

This method uses weighted principal component analysis (WPCA) to fit a R-dimensional hyperplane through the data with initial internal weights all equal. At each iteration the internal weights are recalculated based on the "residuals". If `method=="L1"`, the internal weights are 1 / sum(abs(r) + reps). This is the same as `method=function(r) 1/sum(abs(r)+reps)`. The "residuals" are orthogonal Euclidean distance of the principal components R,R+1,...,K. In each iteration before doing WPCA, the internal weighted are multiplied by the weights given by argument `w`, if specified.

Value

Returns the fit (a `list`) from the last call to `*wpca()` with the additional elements `nbrOfIterations` and `converged`.

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

See Also

Internally `*wpca()` is used for calculating the weighted PCA.

Examples

```
for (zzz in 0) {

# This example requires plot3d() in R.basic [http://www.braju.com/R/]
if (!require(R.basic)) break

# Simulate data from the model y <- a + bx + eps(bx)
x <- rexp(1000)
a <- c(2,15,3)
b <- c(2,3,4)
bx <- outer(b,x)
eps <- apply(bx, MARGIN=2, FUN=function(x) rnorm(length(x), mean=0, sd=0.1*x))
y <- a + bx + eps
y <- t(y)

# Add some outliers by permuting the dimensions for 1/10 of the observations
```

```
idx <- sample(1:nrow(y), size=1/10*nrow(y))
y[idx,] <- y[idx,c(2,3,1)]

# Plot the data with fitted lines at four different view points
opar <- par(mar=c(1,1,1,1)+0.1)
N <- 4
layout(matrix(1:N, nrow=2, byrow=TRUE))
theta <- seq(0,270,length=N)
phi <- rep(20, length.out=N)
xlim <- ylim <- zlim <- c(0,45);
persp <- list();
for (kk in seq(theta)) {
  # Plot the data
  persp[[kk]] <- plot3d(y, theta=theta[kk], phi=phi[kk], xlim=xlim, ylim=ylim, zlim=zlim)
}

# Weights on the observations
# Example a: Equal weights
w <- NULL
# Example b: More weight on the outliers (uncomment to test)
w <- rep(1, length(x)); w[idx] <- 0.8

# ...and show all iterations too with different colors.
maxIter <- c(seq(1,20,length=10),Inf)
col <- topo.colors(length(maxIter))
# Show the fitted value for every iteration
for (ii in seq(along=maxIter)) {
  # Fit a line using IWPCA through data
  fit <- iwpca(y, w=w, maxIter=maxIter[ii], swapDirections=TRUE)

  ymid <- fit$xMean
  d0 <- apply(y, MARGIN=2, FUN=min) - ymid
  d1 <- apply(y, MARGIN=2, FUN=max) - ymid
  b <- fit$vt[1,]
  y0 <- -b * max(abs(d0))
  y1 <-  b * max(abs(d1))
  yline <- matrix(c(y0,y1), nrow=length(b), ncol=2)
  yline <- yline + ymid

  for (kk in seq(theta)) {
    # Set pane to draw in
    par(mfg=c((kk-1) %/% 2, (kk-1) %% 2) + 1);
    # Set the viewpoint of the pane
    options(persp.matrix=persp[[kk]]);

    # Get the first principal component
    points3d(t(ymid), col=col[ii])
    lines3d(t(yline), col=col[ii])

    # Highlight the last one
    if (ii == length(maxIter))
      lines3d(t(yline), col="red", lwd=3)
  }
}

par(opar)
```

```
} # for (zzz in 0)
rm(zzz)
```

`likelihood.smooth.spline`
Calculate the log likelihood of a smoothing spline given the data

---

Description

Calculate the (log) likelihood of a spline given the data used to fit the spline, g. The likelihood consists of two main parts: 1) (weighted) residuals sum of squares, and 2) a penalty term. The penalty term consists of a *smoothing parameter* $lambda$ and a *roughness measure* of the spline $J(g) = \int g''(t)dt$. Hence, the overall log likelihood is

$$\log L(g|x) = (y - g(x))'W(y - g(x)) + \lambda J(g)$$

In addition to the overall likelihood, all its seperate components are also returned.

Note: when fitting a smooth spline with (x, y) values where the x's are *not* unique, `smooth.spline` will replace such (x, y)'s with a new pair (x, y') where y' is a reweighted average on the original y's. It is important to be aware of this. In such cases, the resulting `smooth.spline` object does *not* contain all (x, y)'s and therefore this function will not calculate the weighted residuals sum of square on the original data set, but on the data set with unique x's. See examples below how to calculate the likelihood for the spline with the original data.

Usage

```
## S3 method for class 'smooth.spline':
likelihood(object, x=NULL, y=NULL, w=NULL, base=exp(1), rel.tol=.Machine$double.
```

Arguments

`object`	The smooth.spline object.
`x, y`	The x and y values for which the (weighted) likelihood will be calculated. If `x` is of type `xy.coords` any value of argument `y` will be omitted. If `x==NULL`, the x and y values of the smoothing spline will be used.
`w`	The weights for which the (weighted) likelihood will be calculated. If `NULL`, weights equal to one are assumed.
`base`	The base of the logarithm of the likelihood. If `NULL`, the non-logged likelihood is returned.
`rel.tol`	The relative tolerance used in the call to `integrate`.
`...`	Not used.

Details

The roughness penalty for the smoothing spline, g, fitted from data in the interval $[a, b]$ is defined as

$$J(g) = \int_a^b g''(t)dt$$

which is the same as

$$J(g) = g'(b) - g'(a)$$

The latter is calculated internally by using `predict.smooth.spline`.

Value

Returns the overall (log) likelihood of class `SmoothSplineLikelihood`, a class with the following attributes:

`wrss`	the (weighted) residual sum of square
`penalty`	the penalty which is equal to `-lambda*roughness`.
`lambda`	the smoothing parameter
`roughness`	the value of the roughness functional given the specific smoothing spline and the range of data

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

See Also

`smooth.spline` and `robustSmoothSpline`().

Examples

```
# Define f(x)
f <- expression(0.1*x^4 + 1*x^3 + 2*x^2 + x + 10*sin(2*x))

# Simulate data from this function in the range [a,b]
a <- -2; b <- 5
x <- seq(a, b, length=3000)
y <- eval(f)

# Add some noise to the data
y <- y + rnorm(length(y), 0, 10)

# Plot the function and its second derivative
plot(x,y, type="l", lwd=4)

# Fit a cubic smoothing spline and plot it
g <- smooth.spline(x,y, df=16)
lines(g, col="yellow", lwd=2, lty=2)

# Calculating the (log) likelihood of the fitted spline
l <- likelihood(g)

cat("Log likelihood with unique x values:\n")
print(l)

# Note that this is not the same as the log likelihood of the
# data on the fitted spline iff the x values are non-unique
x[1:5] <- x[1]  # Non-unique x values
g <- smooth.spline(x,y, df=16)
l <- likelihood(g)

cat("\nLog likelihood of the *spline* data set:\n");
print(l)

# In cases with non unique x values one has to proceed as
# below if one want to get the log likelihood for the original
# data.
l <- likelihood(g, x=x, y=y)
cat("\nLog likelihood of the *original* data set:\n");
print(l)
```

medianPolish.matrix

Median polish

Description

Median polish.

Usage

```
## S3 method for class 'matrix':
medianPolish(X, tol=0.01, maxIter=10, na.rm=NA, ..., .addExtra=TRUE)
```

Arguments

`X`	N-times-K `matrix`
`tol`	A `numeric` value greater than zero used as a threshold to identify when the algorithm has converged.
`maxIter`	Maximum number of iterations.
`na.rm`	If `TRUE` (`FALSE`), `NA`s are exclude (not exclude). If `NA`, it is assumed that `X` contains no `NA` values.
`.addExtra`	If `TRUE`, the name of argument `X` is returned and the returned structure is assigned a class. This will make the result compatible what `medpolish` returns.
`...`	Not used.

Details

The implementation of this method give identical estimates as `medpolish`, but is about 3-5 times more efficient when there are no `NA` values.

Value

Returns a named `list` structure with elements:

`overall`	The fitted constant term.
`row`	The fitted row effect.
`col`	The fitted column effect.
`residuals`	The residuals.
`converged`	If `TRUE`, the algorithm converged, otherwise not.

Author(s)

Henrik Bengtsson (http://www.braju.com/R/)

See Also

medpolish.

Examples

```
# Deaths from sport parachuting;  from ABC of EDA, p.224:
deaths <- matrix(c(14,15,14, 7,4,7, 8,2,10, 15,9,10, 0,2,0), ncol=3, byrow=TRUE)
rownames(deaths) <- c("1-24", "25-74", "75-199", "200++", "NA")
colnames(deaths) <- 1973:1975

print(deaths)

mp <- medianPolish(deaths)
mp1 <- medpolish(deaths, trace=FALSE)
print(mp)

ff <- c("overall", "row", "col", "residuals")
stopifnot(all.equal(mp[ff], mp1[ff]))

# Validate decomposition:
stopifnot(all.equal(deaths, mp$overall+outer(mp$row,mp$col,"+")+mp$resid))
```

---

normalizeAffine.matrix

Weighted affine normalization between channels and arrays

Description

Weighted affine normalization between channels and arrays.

This method will both remove curvature in the M vs A plots that are due to an affine transformation of the data. In other words, if there are (small or large) biases in the different (red or green) channels, biases that can be equal too, you will get curvature in the M vs A plots and this type of curvature will be removed by this normalization method.

Moreover, if you normalize all slides at once, this method will also bring the signals on the same scale such that the log-ratios for different slides are comparable. Thus, do not normalize the scale of the log-ratios between slides afterward.

It is recommended to normalize as many slides as possible in one run. The result is that if creating log-ratios between any channels and any slides, they will contain as little curvature as possible.

Furthermore, since the relative scale between any two channels on any two slides will be one if one normalizes all slides (and channels) at once it is possible to add or multiply with the *same* constant to all channels/arrays without introducing curvature. Thus, it is easy to rescale the data afterwards as demonstrated in the example.

Usage

```
## S3 method for class 'matrix':
normalizeAffine(X, weights=NULL, typeOfWeights=c("datapoint"), method="L1", cons
```

Arguments

`X`	An NxK `matrix` (K>=2) where the columns represent the channels, to be normalized.
`weights`	If `NULL`, non-weighted normalization is done. If data-point weights are used, this should be a `vector` of length N of data point weights used when estimating the normalization function.
`typeOfWeights`	A `character` string specifying the type of weights given in argument `weights`.
`method`	A `character` string specifying how the estimates are robustified. See `*iwpca()` for all accepted values.
`constraint`	Constraint making the bias parameters identifiable. See `*fitIWPCA()` for more details.
`satSignal`	Signals equal to or above this threshold will not be used in the fitting.
`...`	Other arguments passed to `*fitIWPCA()` and in turn `*iwpca()`. For example, the weight argument of `*iwpca()`. See also below.
`.fitOnly`	If `TRUE`, the data will not be back-transform.

Details

A line is fitted robustly throught the (y_R, y_G) observations using an iterated re-weighted principal component analysis (IWPCA), which minimized the residuals that are orthogonal to the fitted line. Each observation is down-weighted by the inverse of the absolute residuals, i.e. the fit is done in L_1.

Value

A NxK `matrix` of the normalized channels. The fitted model is returned as attribute `modelFit`.

Negative, non-positive, and saturated values

Affine normalization applies equally well to negative values. Thus, contrary to normalization methods applied to log-ratios, such as curve-fit normalization methods, affine normalization, will not set these to `NA`.

Data points that are saturated in one or more channels are not used to estimate the normalization function, but they are normalized.

Missing values

The estimation of the affine normalization function will only be made based on complete non-saturated observations, i.e. observations that contains no `NA` values nor saturated values as defined by `satSignal`.

Weighted normalization

Each data point/observation, that is, each row in `X`, which is a vector of length K, can be assigned a weight in [0,1] specifying how much it should *affect the fitting of the affine normalization function*. Weights are given by argument `weights`, which should be a `numeric vector` of length N. Regardless of weights, all data points are *normalized* based on the fitted normalization function.

Robustness

By default, the model fit of affine normalization is done in L_1 (`method="L1"`). This way, outliers affect the parameter estimates less than ordinary least-square methods.

For further robustness, downweight outliers such as saturated signals, if possible.

We do not use Tukey's biweight function for reasons similar to those outlined in `*calibrateMultiscan()`.

Using known/previously estimated channel offsets

If the channel offsets can be assumed to be known, then it is possible to fit the affine model with no (zero) offset, which formally is a linear (proportional) model, by specifying argument `center=FALSE`. In order to do this, the channel offsets have to be subtracted from the signals manually before normalizing, e.g. `Xa <- t(t(X)-a)` where `e` is `vector` of length `ncol(X)`. Then normalize by `Xn <- normalizeAffine(Xa, center=FALSE)`. You can assert that the model is fitted without offset by `stopifnot(all(attr(Xn, "modelFit")$adiag == 0))`.

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

References

[1] Henrik Bengtsson and Ola Hössjer, *Methodological Study of Affine Transformations of Gene Expression Data*, Methodological study of affine transformations of gene expression data with proposed robust non-parametric multi-dimensional normalization method, BMC Bioinformatics, 2006, 7:100.

See Also

`*calibrateMultiscan()`.

Examples

```
pathname <- system.file("data-ex", "PMT-RGData.dat", package="aroma.light")
rg <- read.table(pathname, header=TRUE, sep="\t")
nbrOfScans <- max(rg$slide)

rg <- as.list(rg)
for (field in c("R", "G"))
  rg[[field]] <- matrix(as.double(rg[[field]]), ncol=nbrOfScans)
rg$slide <- rg$spot <- NULL
rg <- as.matrix(as.data.frame(rg))
colnames(rg) <- rep(c("R", "G"), each=nbrOfScans)

layout(matrix(c(1,2,0,3,4,0,5,6,7), ncol=3, byrow=TRUE))

rgC <- rg
for (channel in c("R", "G")) {
  sidx <- which(colnames(rg) == channel)
  channelColor <- switch(channel, R="red", G="green");

  # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  # The raw data
  # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  plotMvsAPairs(rg[,sidx])
  title(main=paste("Observed", channel))
  box(col=channelColor)

  # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  # The calibrated data
  # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  rgC[,sidx] <- calibrateMultiscan(rg[,sidx], average=NULL)

  plotMvsAPairs(rgC[,sidx])
  title(main=paste("Calibrated", channel))
  box(col=channelColor)
} # for (channel ...)


# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# The average calibrated data
#
# Note how the red signals are weaker than the green. The reason
# for this can be that the scale factor in the green channel is
# greater than in the red channel, but it can also be that there
# is a remaining relative difference in bias between the green
# and the red channel, a bias that precedes the scanning.
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
rgCA <- rg
for (channel in c("R", "G")) {
  sidx <- which(colnames(rg) == channel)
  rgCA[,sidx] <- calibrateMultiscan(rg[,sidx])
}

rgCAavg <- matrix(NA, nrow=nrow(rgCA), ncol=2)
colnames(rgCAavg) <- c("R", "G");
for (channel in c("R", "G")) {
  sidx <- which(colnames(rg) == channel)
  rgCAavg[,channel] <- apply(rgCA[,sidx], MARGIN=1, FUN=median, na.rm=TRUE);
}

# Add some "fake" outliers
outliers <- 1:600
rgCAavg[outliers,"G"] <- 50000;

plotMvsA(rgCAavg)
title(main="Average calibrated (AC)")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Normalize data
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Weight-down outliers when normalizing
weights <- rep(1, nrow(rgCAavg));
weights[outliers] <- 0.001;

# Affine normalization of channels
rgCANa <- normalizeAffine(rgCAavg, weights=weights)
# It is always ok to rescale the affine normalized data if its
# done on (R,G); not on (A,M)! However, this is only needed for
# esthetic purposes.
```

```
rgCANa <- rgCANa *2^1.4
plotMvsA(rgCANa)
title(main="Normalized AC")

# Curve-fit (lowess) normalization
rgCANlw <- normalizeLowess(rgCAavg, weights=weights)
plotMvsA(rgCANlw, col="orange", add=TRUE)

# Curve-fit (loess) normalization
rgCANl <- normalizeLoess(rgCAavg, weights=weights)
plotMvsA(rgCANl, col="red", add=TRUE)

# Curve-fit (robust spline) normalization
rgCANrs <- normalizeRobustSpline(rgCAavg, weights=weights)
plotMvsA(rgCANrs, col="blue", add=TRUE)

legend(x=0,y=16, legend=c("affine", "lowess", "loess", "r. spline"), pch=19,
       col=c("black", "orange", "red", "blue"), ncol=2, x.intersp=0.3, bty="n")


plotMvsMPairs(cbind(rgCANa, rgCANlw), col="orange", xlab=expression(M[affine]))
title(main="Normalized AC")
plotMvsMPairs(cbind(rgCANa, rgCANl), col="red", add=TRUE)
plotMvsMPairs(cbind(rgCANa, rgCANrs), col="blue", add=TRUE)
abline(a=0, b=1, lty=2)
legend(x=-6,y=6, legend=c("lowess", "loess", "r. spline"), pch=19,
       col=c("orange", "red", "blue"), ncol=2, x.intersp=0.3, bty="n")
```

normalizeAverage.matrix

Rescales channel vectors to get the same average

Description

Rescales channel vectors to get the same average.

Usage

```
## S3 method for class 'matrix':
normalizeAverage(x, baseline=1, avg=median, targetAvg=2200, ...)
```

Arguments

`x`	A `numeric` NxK `matrix` (or `list` of length K).
`baseline`	An `integer` in [1,K] specifying which channel should be the baseline.
`avg`	A `function` for calculating the average of one channel.
`targetAvg`	The average that each channel should have afterwards. If `NULL`, the baseline column sets the target average.
`...`	Additional arguments passed to the `avg` `function`.

Value

Returns a normalized numeric NxK matrix (or list of length K).

Author(s)

Henrik Bengtsson (http://www.braju.com/R/)

---

normalizeCurveFit.matrix

Weighted curve-fit normalization between a pair of channels

Description

Weighted curve-fit normalization between a pair of channels.

This method will estimate a smooth function of the dependency between the log-ratios and the log-intensity of the two channels and then correct the log-ratios (only) in order to remove the dependency. This is method is also known as *intensity-dependent* or *lowess normalization*.

The curve-fit methods are by nature limited to paired-channel data. There exist at least one method trying to overcome this limitation, namely the cyclic-lowess [1], which applies the paired curve-fit method iteratively over all pairs of channels/arrays. Cyclic-lowess is not implented here.

We recommend that affine normalization [2] is used instead of curve-fit normalization.

Usage

```
## S3 method for class 'matrix':
normalizeCurveFit(X, weights=NULL, typeOfWeights=c("datapoint"), method=c("loess
```

Arguments

`X`	An Nx2 `matrix` where the columns represent the two channels to be normalized.
`weights`	If `NULL`, non-weighted normalization is done. If data-point weights are used, this should be a `vector` of length N of data point weights used when estimating the normalization function.
`typeOfWeights`	A `character` string specifying the type of weights given in argument `weights`.
`method`	`character` string specifying which method to use when fitting the intensity-dependent function. Supported methods: `"loess"` (better than lowess), `"lowess"` (classic; supports only zero-one weights), `"spline"` (more robust than lowess at lower and upper intensities; supports only zero-one weights), `"robustSpline"` (better than spline).
`bandwidth`	A `double` value specifying the bandwidth of the estimator used.
`satSignal`	Signals equal to or above this threshold will not be used in the fitting.
`...`	Not used.

Details

A smooth function $c(A)$ is fitted throught data in (A, M), where $M = log_2(y_2/y_1)$ and $A = 1/2 * log_2(y_2 * y_1)$. Data is normalized by $M <- M - c(A)$.

Loess is by far the slowest method of the four, then lowess, and then robust spline, which iteratively calls the spline method.

Value

A Nx2 `matrix` of the normalized two channels. The fitted model is returned as attribute `modelFit`.

Negative, non-positive, and saturated values

Non-positive values are set to not-a-number (`NaN`). Data points that are saturated in one or more channels are not used to estimate the normalization function, but they are normalized.

Missing values

The estimation of the affine normalization function will only be made based on complete non-saturated observations, i.e. observations that contains no `NA` values nor saturated values as defined by `satSignal`.

Weighted normalization

Each data point, that is, each row in `X`, which is a vector of length 2, can be assigned a weight in [0,1] specifying how much it should *affect the fitting of the affine normalization function*. Weights are given by argument `weights`, which should be a `numeric` `vector` of length N. Regardless of weights, all data points are *normalized* based on the fitted normalization function.

Note that the lowess and the spline method only support zero-one {0,1} weights. For such methods, all weights that are less than a half are set to zero.

Details on loess

For `loess`, the arguments `family="symmetric"`, `degree=1`, `span=3/4`, `control=loess.control(trac` `iterations=5`, `surface="direct")` are used.

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

References

[1] M. Åstrand, Contrast Normalization of Oligonucleotide Arrays, Journal Computational Biology, 2003, 10, 95-102.
[2] Henrik Bengtsson and Ola Hössjer, *Methodological Study of Affine Transformations of Gene Expression Data*, Methodological study of affine transformations of gene expression data with proposed robust non-parametric multi-dimensional normalization method, BMC Bioinformatics, 2006, 7:100.

See Also

`*normalizeAffine()`.

Examples

```
 pathname <- system.file("data-ex", "PMT-RGData.dat", package="aroma.light")
rg <- read.table(pathname, header=TRUE, sep="\t")
nbrOfScans <- max(rg$slide)

rg <- as.list(rg)
for (field in c("R", "G"))
  rg[[field]] <- matrix(as.double(rg[[field]]), ncol=nbrOfScans)
rg$slide <- rg$spot <- NULL
rg <- as.matrix(as.data.frame(rg))
colnames(rg) <- rep(c("R", "G"), each=nbrOfScans)

layout(matrix(c(1,2,0,3,4,0,5,6,7), ncol=3, byrow=TRUE))

rgC <- rg
for (channel in c("R", "G")) {
  sidx <- which(colnames(rg) == channel)
  channelColor <- switch(channel, R="red", G="green");

  # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  # The raw data
  # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  plotMvsAPairs(rg[,sidx])
  title(main=paste("Observed", channel))
  box(col=channelColor)

  # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  # The calibrated data
  # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  rgC[,sidx] <- calibrateMultiscan(rg[,sidx], average=NULL)

  plotMvsAPairs(rgC[,sidx])
  title(main=paste("Calibrated", channel))
  box(col=channelColor)
} # for (channel ...)


# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# The average calibrated data
#
# Note how the red signals are weaker than the green. The reason
# for this can be that the scale factor in the green channel is
# greater than in the red channel, but it can also be that there
# is a remaining relative difference in bias between the green
# and the red channel, a bias that precedes the scanning.
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
rgCA <- rg
for (channel in c("R", "G")) {
  sidx <- which(colnames(rg) == channel)
  rgCA[,sidx] <- calibrateMultiscan(rg[,sidx])
}

rgCAavg <- matrix(NA, nrow=nrow(rgCA), ncol=2)
colnames(rgCAavg) <- c("R", "G");
for (channel in c("R", "G")) {
  sidx <- which(colnames(rg) == channel)
  rgCAavg[,channel] <- apply(rgCA[,sidx], MARGIN=1, FUN=median, na.rm=TRUE);
}

# Add some "fake" outliers
outliers <- 1:600
rgCAavg[outliers,"G"] <- 50000;

plotMvsA(rgCAavg)
title(main="Average calibrated (AC)")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Normalize data
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Weight-down outliers when normalizing
weights <- rep(1, nrow(rgCAavg));
weights[outliers] <- 0.001;

# Affine normalization of channels
rgCANa <- normalizeAffine(rgCAavg, weights=weights)
# It is always ok to rescale the affine normalized data if its
# done on (R,G); not on (A,M)! However, this is only needed for
# esthetic purposes.
rgCANa <- rgCANa *2^1.4
plotMvsA(rgCANa)
title(main="Normalized AC")

# Curve-fit (lowess) normalization
rgCANlw <- normalizeLowess(rgCAavg, weights=weights)
plotMvsA(rgCANlw, col="orange", add=TRUE)

# Curve-fit (loess) normalization
rgCANl <- normalizeLoess(rgCAavg, weights=weights)
plotMvsA(rgCANl, col="red", add=TRUE)

# Curve-fit (robust spline) normalization
rgCANrs <- normalizeRobustSpline(rgCAavg, weights=weights)
plotMvsA(rgCANrs, col="blue", add=TRUE)

legend(x=0,y=16, legend=c("affine", "lowess", "loess", "r. spline"), pch=19,
       col=c("black", "orange", "red", "blue"), ncol=2, x.intersp=0.3, bty="n")


plotMvsMPairs(cbind(rgCANa, rgCANlw), col="orange", xlab=expression(M[affine]))
title(main="Normalized AC")
plotMvsMPairs(cbind(rgCANa, rgCANl), col="red", add=TRUE)
plotMvsMPairs(cbind(rgCANa, rgCANrs), col="blue", add=TRUE)
abline(a=0, b=1, lty=2)
legend(x=-6,y=6, legend=c("lowess", "loess", "r. spline"), pch=19,
       col=c("orange", "red", "blue"), ncol=2, x.intersp=0.3, bty="n")
```

`normalizeDifferencesToAverage.list` *Rescales channel vectors to get the same average*

Description

Rescales channel vectors to get the same average.

Usage

```
## S3 method for class 'list':
normalizeDifferencesToAverage(x, baseline=1, FUN=median, ...)
```

Arguments

`x`	A `numeric` `list` of length K.
`baseline`	An `integer` in [1,K] specifying which channel should be the baseline. The baseline channel will be almost unchanged. If `NULL`, the channels will be shifted towards median of them all.
`FUN`	A `function` for calculating the average of one channel.
`...`	Additional arguments passed to the `avg` `function`.

Value

Returns a normalized `list` of length K.

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

Examples

```
# Simulate three shifted tracks of different lengths with same profiles
ns <- c(A=2, B=1, C=0.25)*1000;
xx <- lapply(ns, FUN=function(n) { seq(from=1, to=max(ns), length.out=n) });
zz <- mapply(seq(along=ns), ns, FUN=function(z,n) rep(z,n));

yy <- list(
  A = rnorm(ns["A"], mean=0, sd=0.5),
  B = rnorm(ns["B"], mean=5, sd=0.4),
  C = rnorm(ns["C"], mean=-5, sd=1.1)
);
yy <- lapply(yy, FUN=function(y) {
  n <- length(y);
  y[1:(n/2)] <- y[1:(n/2)] + 2;
  y[1:(n/4)] <- y[1:(n/4)] - 4;
  y;
});

# Shift all tracks toward the first track
yyN <- normalizeDifferencesToAverage(yy, baseline=1);

# The baseline channel is not changed
stopifnot(identical(yy[[1]], yyN[[1]]));

# Get the estimated parameters
fit <- attr(yyN, "fit");

# Plot the tracks
```

```
layout(matrix(1:2, ncol=1));
x <- unlist(xx);
col <- unlist(zz);
y <- unlist(yy);
yN <- unlist(yyN);
plot(x, y, col=col, ylim=c(-10,10));
plot(x, yN, col=col, ylim=c(-10,10));
```

`normalizeFragmentLength` *Normalizes signals for PCR fragment-length effects*

Description

Normalizes signals for PCR fragment-length effects. Some or all signals are used to estimated the normalization function. All signals are normalized.

Usage

```
## Default S3 method:
normalizeFragmentLength(y, fragmentLengths, targetFcns=NULL, subsetToFit=NULL, c
```

Arguments

`y`	A `numeric` `vector` of length K of signals to be normalized across E enzymes.
`fragmentLengths`	An `integer` KxE `matrix` of fragment lengths.
`targetFcns`	An optional `list` of E `function`s; one per enzyme. If `NULL`, the data is normalized to have constant fragment-length effects (all equal to zero on the log-scale).
`subsetToFit`	The subset of data points used to fit the normalization function. If `NULL`, all data points are considered.
`onMissing`	Specifies how data points for which there is no fragment length is normalized. If `"ignore"`, the values are not modified. If `"median"`, the values are updated to have the same robust average as the other data points.
`.isLogged`	A `logical`.
`...`	Additional arguments passed to `lowess`.
`.returnFit`	A `logical`.

Value

Returns a `numeric` `vector` of the normalized signals.

Multi-enzyme normalization

It is assumed that the fragment-length effects from multiple enzymes added (with equal weights) on the intensity scale. The fragment-length effects are fitted for each enzyme separately based on units that are exclusively for that enzyme. *If there are no or very such units for an enzyme, the assumptions of the model are not met and the fit will fail with an error.* Then, from the above single-enzyme fits the average effect across enzymes is the calculated for each unit that is on multiple enzymes.

Target functions

It is possible to specify custom target function effects for each enzyme via argument `targetFcns`. This argument has to be a `list` containing one `function` per enzyme and ordered in the same order as the enzyme are in the columns of argument `fragmentLengths`. For instance, if one wish to normalize the signals such that their mean signal as a function of fragment length effect is contantly equal to 2200 (or the intensity scale), the use `targetFcns=function(fl, ...) log2(2200)` which completely ignores fragment-length argument 'fl' and always returns a constant. If two enzymes are used, then use `targetFcns=rep(list(function(fl, ...) log2(2200)), 2)`.

Note, if `targetFcns` is `NULL`, this corresponds to `targetFcns=rep(list(function(fl, ...) 0), ncol(fragmentLengths))`.

Alternatively, if one wants to only apply minimial corrections to the signals, then one can normalize toward target functions that correspond to the fragment-length effect of the average array.

Author(s)

Henrik Bengtsson (http://www.braju.com/R/)

References

[1] H. Bengtsson, R. Irizarry, B. Carvalho, and T. Speed, *Estimation and assessment of raw copy numbers at the single locus level*, Bioinformatics, 2008.

Examples

```
   # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Example 1: Single-enzyme fragment-length normalization of 6 arrays
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Number samples
I <- 9;

# Number of loci
J <- 1000;

# Fragment lengths
fl <- seq(from=100, to=1000, length.out=J);

# Simulate data points with unknown fragment lengths
hasUnknownFL <- seq(from=1, to=J, by=50);
fl[hasUnknownFL] <- NA;

# Simulate data
y <- matrix(0, nrow=J, ncol=I);
maxY <- 12;
for (kk in 1:I) {
  k <- runif(n=1, min=3, max=5);
  mu <- function(fl) {
    mu <- rep(maxY, length(fl));
    ok <- !is.na(fl);
    mu[ok] <- mu[ok] - fl[ok]^{1/k};
    mu;
  }
  eps <- rnorm(J, mean=0, sd=1);
  y[,kk] <- mu(fl) + eps;
}

# Normalize data (to a zero baseline)
yN <- apply(y, MARGIN=2, FUN=function(y) {
  normalizeFragmentLength(y, fragmentLengths=fl, onMissing="median");
})

# The correction factors
rho <- y-yN;
print(summary(rho));
# The correction for units with unknown fragment lengths
# equals the median correction factor of all other units
print(summary(rho[hasUnknownFL,]));

# Plot raw data
layout(matrix(1:9, ncol=3));
xlim <- c(0,max(fl, na.rm=TRUE));
ylim <- c(0,max(y, na.rm=TRUE));
xlab <- "Fragment length";
ylab <- expression(log2(theta));
for (kk in 1:I) {
  plot(fl, y[,kk], xlim=xlim, ylim=ylim, xlab=xlab, ylab=ylab);
  ok <- (is.finite(fl) & is.finite(y[,kk]));
  lines(lowess(fl[ok], y[ok,kk]), col="red", lwd=2);
}

# Plot normalized data
layout(matrix(1:9, ncol=3));
ylim <- c(-1,1)*max(y, na.rm=TRUE)/2;
for (kk in 1:I) {
  plot(fl, yN[,kk], xlim=xlim, ylim=ylim, xlab=xlab, ylab=ylab);
  ok <- (is.finite(fl) & is.finite(y[,kk]));
  lines(lowess(fl[ok], yN[ok,kk]), col="blue", lwd=2);
}


   # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Example 2: Two-enzyme fragment-length normalization of 6 arrays
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
set.seed(0xbeef);

# Number samples
I <- 5;

# Number of loci
J <- 3000;

# Fragment lengths (two enzymes)
fl <- matrix(0, nrow=J, ncol=2);
fl[,1] <- seq(from=100, to=1000, length.out=J);
fl[,2] <- seq(from=1000, to=100, length.out=J);

# Let 1/2 of the units be on both enzymes
fl[seq(from=1, to=J, by=4),1] <- NA;
fl[seq(from=2, to=J, by=4),2] <- NA;
```

```
# Let some have unknown fragment lengths
hasUnknownFL <- seq(from=1, to=J, by=15);
fl[hasUnknownFL,] <- NA;

# Sty/Nsp mixing proportions:
rho <- rep(1, I);
rho[1] <- 1/3;  # Less Sty in 1st sample
rho[3] <- 3/2;  # More Sty in 3rd sample


# Simulate data
z <- array(0, dim=c(J,2,I));
maxLog2Theta <- 12;
for (ii in 1:I) {
  # Common effect for both enzymes
  mu <- function(fl) {
    k <- runif(n=1, min=3, max=5);
    mu <- rep(maxLog2Theta, length(fl));
    ok <- is.finite(fl);
    mu[ok] <- mu[ok] - fl[ok]^{1/k};
    mu;
  }

  # Calculate the effect for each data point
  for (ee in 1:2) {
    z[,ee,ii] <- mu(fl[,ee]);
  }

  # Update the Sty/Nsp mixing proportions
  ee <- 2;
  z[,ee,ii] <- rho[ii]*z[,ee,ii];

  # Add random errors
  for (ee in 1:2) {
    eps <- rnorm(J, mean=0, sd=1/sqrt(2));
    z[,ee,ii] <- z[,ee,ii] + eps;
  }
}


hasFl <- is.finite(fl);

unitSets <- list(
  nsp  = which( hasFl[,1] & !hasFl[,2]),
  sty  = which(!hasFl[,1] &  hasFl[,2]),
  both = which( hasFl[,1] &  hasFl[,2]),
  none = which(!hasFl[,1] & !hasFl[,2])
)

# The observed data is a mix of two enzymes
theta <- matrix(NA, nrow=J, ncol=I);

# Single-enzyme units
for (ee in 1:2) {
  uu <- unitSets[[ee]];
  theta[uu,] <- 2^z[uu,ee,];
}
```

```
# Both-enzyme units (sum on intensity scale)
uu <- unitSets$both;
theta[uu,] <- (2^z[uu,1,]+2^z[uu,2,])/2;

# Missing units (sample from the others)
uu <- unitSets$none;
theta[uu,] <- apply(theta, MARGIN=2, sample, size=length(uu))

# Calculate target array
thetaT <- rowMeans(theta, na.rm=TRUE);
targetFcns <- list();
for (ee in 1:2) {
  uu <- unitSets[[ee]];
  fit <- lowess(fl[uu,ee], log2(thetaT[uu]));
  class(fit) <- "lowess";
  targetFcns[[ee]] <- function(fl, ...) {
    predict(fit, newdata=fl);
  }
}


# Fit model only to a subset of the data
subsetToFit <- setdiff(1:J, seq(from=1, to=J, by=10))

# Normalize data (to a target baseline)
thetaN <- matrix(NA, nrow=J, ncol=I);
fits <- vector("list", I);
for (ii in 1:I) {
  lthetaNi <- normalizeFragmentLength(log2(theta[,ii]), targetFcns=targetFcns,
                    fragmentLengths=fl, onMissing="median",
                    subsetToFit=subsetToFit, .returnFit=TRUE);
  fits[[ii]] <- attr(lthetaNi, "modelFit");
  thetaN[,ii] <- 2^lthetaNi;
}


# Plot raw data
xlim <- c(0, max(fl, na.rm=TRUE));
ylim <- c(0, max(log2(theta), na.rm=TRUE));
Mlim <- c(-1,1)*4;
xlab <- "Fragment length";
ylab <- expression(log2(theta));
Mlab <- expression(M==log[2](theta/theta[R]));

layout(matrix(1:(3*I), ncol=I, byrow=TRUE));
for (ii in 1:I) {
  plot(NA, xlim=xlim, ylim=ylim, xlab=xlab, ylab=ylab, main="raw");

  # Single-enzyme units
  for (ee in 1:2) {
    # The raw data
    uu <- unitSets[[ee]];
    points(fl[uu,ee], log2(theta[uu,ii]), col=ee+1);
  }

  # Both-enzyme units (use fragment-length for enzyme #1)
  uu <- unitSets$both;
  points(fl[uu,1], log2(theta[uu,ii]), col=3+1);

  for (ee in 1:2) {
    # The true effects
    uu <- unitSets[[ee]];
    lines(lowess(fl[uu,ee], log2(theta[uu,ii])), col="black", lwd=4, lty=3);

    # The estimated effects
    fit <- fits[[ii]][[ee]]$fit;
    lines(fit, col="orange", lwd=3);

    muT <- targetFcns[[ee]](fl[uu,ee]);
    lines(fl[uu,ee], muT, col="cyan", lwd=1);
  }
}

# Calculate log-ratios
thetaR <- rowMeans(thetaN, na.rm=TRUE);
M <- log2(thetaN/thetaR);

# Plot normalized data
for (ii in 1:I) {
  plot(NA, xlim=xlim, ylim=Mlim, xlab=xlab, ylab=Mlab, main="normalized");
  # Single-enzyme units
  for (ee in 1:2) {
    # The normalized data
    uu <- unitSets[[ee]];
    points(fl[uu,ee], M[uu,ii], col=ee+1);
  }
  # Both-enzyme units (use fragment-length for enzyme #1)
  uu <- unitSets$both;
  points(fl[uu,1], M[uu,ii], col=3+1);
}

ylim <- c(0,1.5);
for (ii in 1:I) {
  data <- list();
  for (ee in 1:2) {
    # The normalized data
    uu <- unitSets[[ee]];
    data[[ee]] <- M[uu,ii];
  }
  uu <- unitSets$both;
  if (length(uu) > 0)
    data[[3]] <- M[uu,ii];

  uu <- unitSets$none;
  if (length(uu) > 0)
    data[[4]] <- M[uu,ii];

  cols <- seq(along=data)+1;
  plotDensity(data, col=cols, xlim=Mlim, xlab=Mlab, main="normalized");

  abline(v=0, lty=2);
}
```

normalizeQuantileRank.list

Normalizes the empirical distribution of a set of samples to a target distribution

Description

Normalizes the empirical distribution of a set of samples to a target distribution. The samples may differ in size.

Usage

```
## S3 method for class 'list':
normalizeQuantileRank(X, xTarget=NULL, ...)
```

Arguments

`X`	a `list` with `numeric` `vector`s. The `vector`s may be of different lengths.
`xTarget`	The target empirical distribution. If `NULL`, the target distribution is calculated as the average empirical distribution of the samples.
`...`	Passed to `normalizeQuantileRank.numeric`().

Value

Returns a `list` of normalized `numeric` `vector` of the same lengths as the corresponding ones in the input matrix.

Missing values

Missing values are excluded. Values that are `NA` remain `NA` after normalization. No new `NA`s are introduced.

Author(s)

Adopted from Gordon Smyth (`http://www.statsci.org/`) in 2002 \& 2006. Original code by Ben Bolstad at Statistics Department, University of California.

See Also

The target empirical distribution is calculated as the average using `*averageQuantile()`. Each `vector` is normalized toward this target disribution using `normalizeQuantileRank.numeric`(). `*normalizeQuantileSpline()`.

Examples

```
# Simulate ten samples of different lengths
N <- 10000
X <- list()
for (kk in 1:8) {
  rfcn <- list(rnorm, rgamma)[[sample(2, size=1)]]
  size <- runif(1, min=0.3, max=1)
  a <- rgamma(1, shape=20, rate=10)
  b <- rgamma(1, shape=10, rate=10)
  values <- rfcn(size*N, a, b)

  # "Censor" values
  values[values < 0 | values > 8] <- NA

  X[[kk]] <- values
}

# Add 20% missing values
X <- lapply(X, FUN=function(x) {
  x[sample(length(x), size=0.20*length(x))] <- NA;
  x
})

# Normalize quantiles
Xn <- normalizeQuantile(X)

# Plot the data
layout(matrix(1:2, ncol=1))
xlim <- range(X, na.rm=TRUE);
plotDensity(X, lwd=2, xlim=xlim, main="The original distributions")
plotDensity(Xn, lwd=2, xlim=xlim, main="The normalized distributions")
```

`normalizeQuantileRank.matrix` *Weighted sample quantile normalization*

Description

Normalizes channels so they all have the same average sample distributions.

The average sample distribution is calculated either robustly or not by utilizing either `weightedMedian()` or `weighted.mean()`. A weighted method is used if any of the weights are different from one.

Usage

```
## S3 method for class 'matrix':
normalizeQuantileRank(X, ties=FALSE, robust=FALSE, weights=NULL, typeOfWeights=c
```

Arguments

`X`	a numerical NxK `matrix` with the K columns representing the channels and the N rows representing the data points.
`robust`	If `TRUE`, the (weighted) median function is used for calculating the average sample distribution, otherwise the (weighted) mean function is used.

`ties`	Should ties be specially treated or not?
`weights`	If `NULL`, non-weighted normalization is done. If channel weights, this should be a `vector` of length K specifying the weights for each channel. If signal weights, it should be an NxK `matrix` specifying the weights for each signal.
`typeOfWeights`	A `character` string specifying the type of weights given in argument `weights`.
`...`	Not used.

Value

Returns an NxK `matrix`.

Missing values

Missing values are excluded when estimating the "common" (the baseline) distribution. Values that are `NA` before remain `NA`. No new `NA`s are introduced.

Weights

Currently only channel weights are support due to the way quantile normalization is done. If signal weights are given, channel weights are calculated from these by taking the mean of the signal weights in each channel.

Author(s)

Adopted from Gordon Smyth (`http://www.statsci.org/`) in 2002 \& 2006. Original code by Ben Bolstad at Statistics Department, University of California. Support for calculating the average sample distribution using (weighted) mean or median was added by Henrik Bengtsson (`http://www.braju.com/R/`).

See Also

`median`, `weightedMedian`(), `mean`() and `weighted.mean`. `*normalizeQuantileSpline()`.

Examples

```
# Simulate three samples with on average 20% missing values
N <- 10000
X <- cbind(rnorm(N, mean=3, sd=1),
           rnorm(N, mean=4, sd=2),
           rgamma(N, shape=2, rate=1))
X[sample(3*N, size=0.20*3*N)] <- NA

# Normalize quantiles
Xn <- normalizeQuantile(X)

# Plot the data
layout(matrix(1:2, ncol=1))
xlim <- range(X, Xn, na.rm=TRUE);
plotDensity(X, lwd=2, xlim=xlim, main="The three original distributions")
plotDensity(Xn, lwd=2, xlim=xlim, main="The three normalized distributions")
```

`normalizeQuantileRank.numeric`

Normalizes the empirical distribution of a single sample to a target distribution

Description

Normalizes the empirical distribution of a single sample to a target distribution.

Usage

```
## S3 method for class 'numeric':
normalizeQuantileRank(x, xTarget, ties=FALSE, ...)
```

Arguments

`x`	a `numeric` `vector` of length N.
`xTarget`	a *sorted* `numeric` `vector` of length M.
`ties`	Should ties in `x` be treated with care or not? For more details, see "limma:normalizeQuantiles".
`...`	Not used.

Value

Returns a `numeric` `vector` of length N.

Missing values

It is only the empirical distribution of the non-missing values that is normalized to the target distribution. All `NA` values remain `NA` after normalization. No new `NA`s are introduced.

Author(s)

Adopted from Gordon Smyth (`http://www.statsci.org/`) in 2002 \& 2006. Original code by Ben Bolstad at Statistics Department, University of California.

See Also

To calculate a target distribution from a set of samples, see `averageQuantile.list`(). This method is used by `normalizeQuantileRank.list`(). `*normalizeQuantileSpline()`.

normalizeQuantileSpline.list

Normalizes the empirical distribution of a set of samples to a target distribution

Description

Normalizes the empirical distribution of a set of samples to a target distribution. The samples may differ in size.

Usage

```
## S3 method for class 'list':
normalizeQuantileSpline(X, xTarget=NULL, ...)
```

Arguments

`X`	a `list` with `numeric` `vector`s. The `vector`s may be of different lengths.
`xTarget`	The target empirical distribution. If `NULL`, the target distribution is calculated as the average empirical distribution of the samples.
`...`	Passed to `normalizeQuantileSpline.numeric`().

Value

Returns a `list` of normalized `numeric` `vector` of the same lengths as the corresponding ones in the input matrix.

Missing values

Missing values are excluded. Values that are `NA` remain `NA` after normalization. No new `NA`s are introduced.

Author(s)

Henrik Bengtsson, Statistics Department, University of California at Berkeley.

See Also

The target empirical distribution is calculated as the average using `*averageQuantile()`. Each `vector` is normalized toward this target disribution using `normalizeQuantileSpline.numeric`(). `*normalizeQuantileRank()`.

normalizeQuantileSpline.matrix

Weighted sample quantile normalization

Description

Normalizes channels so they all have the same average sample distributions.

Usage

```
## S3 method for class 'matrix':
normalizeQuantileSpline(X, xTarget, ...)
```

Arguments

X	A numeric NxK matrix with the K columns representing the channels and the N rows representing the data points.
xTarget	A numeric vector of length N.
...	Additional arguments passed to normalizeQuantileSpline.numeric().

Value

Returns an NxK matrix.

Missing values

Both argument X and xTarget may contain non-finite values. These values do not affect the estimation of the normalization function. Non-finite values in X, remain in the output.

Author(s)

Henrik Bengtsson (http://www.braju.com/R/)

References

[1] H. Bengtsson, R. Irizarry, B. Carvalho, and T. Speed, *Estimation and assessment of raw copy numbers at the single locus level*, Bioinformatics, 2008.

See Also

Internally normalizeQuantileSpline.numeric() is used. *normalizeQuantileRank().

Examples

```
# Simulate three samples with on average 20% missing values
N <- 10000
X <- cbind(rnorm(N, mean=3, sd=1),
           rnorm(N, mean=4, sd=2),
           rgamma(N, shape=2, rate=1))
X[sample(3*N, size=0.20*3*N)] <- NA
```

```
# Plot the data
layout(matrix(c(1,0,2:5), ncol=2, byrow=TRUE))
xlim <- range(X, na.rm=TRUE);
plotDensity(X, lwd=2, xlim=xlim, main="The three original distributions")

Xn <- normalizeQuantile(X)
plotDensity(Xn, lwd=2, xlim=xlim, main="The three normalized distributions")
plotXYCurve(X, Xn, xlim=xlim, main="The three normalized distributions")

Xn2 <- normalizeQuantileSpline(X, xTarget=Xn[,1], spar=0.99)
plotDensity(Xn2, lwd=2, xlim=xlim, main="The three normalized distributions")
plotXYCurve(X, Xn2, xlim=xlim, main="The three normalized distributions")
```

`normalizeQuantileSpline.numeric`

Normalizes the empirical distribution of a single sample to a target distribution

---

Description

Normalizes the empirical distribution of a single sample to a target distribution.

Usage

```
## S3 method for class 'numeric':
normalizeQuantileSpline(x, w=NULL, xTarget, sortTarget=TRUE, ..., robust=TRUE)
```

Arguments

`x`	a `numeric` `vector` of length N.
`w`	an optional `numeric` `vector` of length N of weights.
`xTarget`	a `numeric` `vector` of length N.
`sortTarget`	If `TRUE`, argument `xTarget` is sorted.
`...`	Arguments passed to (`smooth.spline` or `robustSmoothSpline`), e.g. `w` for weights.
`robust`	If `TRUE`, the normalization function is estimated robustly.

Value

Returns a `numeric` `vector` of length N.

Missing values

Both argument `X` and `xTarget` may contain non-finite values. These values do not affect the estimation of the normalization function. Non-finite values in `X`, remain in the output.

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

References

[1] H. Bengtsson, R. Irizarry, B. Carvalho, and T. Speed, *Estimation and assessment of raw copy numbers at the single locus level*, Bioinformatics, 2008.

See Also

Internally either `robustSmoothSpline` or `smooth.spline` is used. `normalizeQuantileSpline.matrix`
`*normalizeQuantileRank()`.

`normalizeTumorBoost.numeric`
Normalizes allele B fractions for a tumor given a match normal

---

Description

TumorBoost [1] is a normalization method that normalizes the allele B fractions of a tumor sample given the allele B fractions and genotypes of a matched normal. The method is a single-sample (single-pair) method. It does not require total copy-number estimates. The normalization is done such that the total copy number is unchanged afterwards.

Usage

```
## S3 method for class 'numeric':
normalizeTumorBoost(betaT, betaN, muN=callNaiveGenotypes(betaN), flavor=c("v4",
```

Arguments

`betaT, betaN`	Two `numeric` `vector`s each of length J with tumor and normal allele B fractions, respectively.
`muN`	An optional `vector` of length J containing normal genotypes calls in (0,1/2,1,`NA`) for (AA,AB,BB).
`flavor`	A `character` string specifying the type of correction applied.
`preserveScale`	If `TRUE`, SNPs that are heterozygous in the matched normal are corrected for signal compression using an estimate of signal compression based on the amount of correction performed by TumorBoost on SNPs that are homozygous in the matched normal.
`...`	Argument passed to `callNaiveGenotypes`(), if called.

Details

Allele B fractions are defined as the ratio between the allele B signal and the sum of both (all) allele signals at the same locus. Allele B fractions are typically within [0,1], but may have a slightly wider support due to for instance negative noise. This is typically also the case for the returned normalized allele B fractions.

Value

Returns a `numeric` `vector` of length J containing the normalized allele B fractions for the tumor. Attribute `modelFit` is a `list` containing model fit parameters.

Flavors

This method provides a few different "flavors" for normalizing the data. The following values of argument `flavor` are accepted:

- v4: (default) The TumorBoost method, i.e. Eqns. (8)-(9) in [1].
- v3: Eqn (9) in [1] is applied to both heterozygous and homozygous SNPs, which effectly is v4 where the normalized allele B fractions for homozygous SNPs becomes 0 and 1.
- v2: ...
- v1: TumorBoost where correction factor is force to one, i.e. $\eta_j = 1$. As explained in [1], this is a suboptimal normalization method. See also the discussion in the paragraph following Eqn (12) in [1].

Preserving scale

Allele B fractions are more or less compressed toward a half, e.g. the signals for homozygous SNPs are slightly away from zero and one. The TumorBoost method decreases the correlation in allele B fractions between the tumor and the normal *conditioned on the genotype*. What it does not control for is the mean level of the allele B fraction *conditioned on the genotype*.

By design, most flavors of the method will correct the homozygous SNPs such that their mean levels get close to the expected zero and one levels. However, the heterozygous SNPs will typically keep the same mean levels as before. One possibility is to adjust the signals such as the mean levels of the heterozygous SNPs relative to that of the homozygous SNPs is the same after as before the normalization.

If argument `preserveScale=TRUE`, then SNPs that are heterozygous (in the matched normal) are corrected for signal compression using an estimate of signal compression based on the amount of correction performed by TumorBoost on SNPs that are homozygous (in the matched normal).

The option of preserving the scale is *not* discussed in the TumorBoost paper [1].

Author(s)

Henrik Bengtsson and Pierre Neuvial

References

[1] H. Bengtsson, P. Neuvial & T.P. Speed, *TumorBoost: Normalization of allele-specific tumor copy numbers from a single pair of tumor-normal genotyping microarrays*, BMC Bioinformatics, 2010, 11:245. [PMID 20462408]

Examples

```
library(R.utils)

# Load data
pathname <- system.file("data-ex/TumorBoost,fracB,exampleData.Rbin", package="aroma.light
data <- loadObject(pathname)
attachLocally(data)
pos <- position/1e6
muN <- genotypeN

layout(matrix(1:4, ncol=1))
par(mar=c(2.5,4,0.5,1)+0.1)
```

```
ylim <- c(-0.05, 1.05)
col <- rep("#999999", length(muN))
col[muN == 1/2] <- "#000000"

# Allele B fractions for the normal sample
plot(pos, betaN, col=col, ylim=ylim)

# Allele B fractions for the tumor sample
plot(pos, betaT, col=col, ylim=ylim)

# TumorBoost w/ naive genotype calls
betaTN <- normalizeTumorBoost(betaT=betaT, betaN=betaN)
plot(pos, betaTN, col=col, ylim=ylim)

# TumorBoost w/ external multi-sample genotype calls
betaTNx <- normalizeTumorBoost(betaT=betaT, betaN=betaN, muN=muN)
plot(pos, betaTNx, col=col, ylim=ylim)
```

`plotDensity.list` *Plots density distributions for a set of vector*

Description

Plots density distributions for a set of vector.

Usage

```
## S3 method for class 'list':
plotDensity(X, xlim=NULL, ylim=NULL, xlab=NULL, ylab="density (integrates to one
```

Arguments

`X`	A single of `list` of `numeric` `vector`s, a `numeric` `matrix`, or a `numeric` `data.frame`.
`xlim,ylim`	`character` `vector` of length 2. The x and y limits.
`xlab,ylab`	`character` string for labels on x and y axis.
`col`	The color(s) of the curves.
`lty`	The types of curves.
`lwd`	The width of curves.
`...`	Additional arguments passed to `density`, `plot`, and `lines`.
`add`	If `TRUE`, the curves are plotted in the current plot, otherwise a new is created.

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

plotMvsA.matrix *Plot log-ratios vs log-intensities*

Description

Plot log-ratios vs log-intensities.

Usage

```
## S3 method for class 'matrix':
plotMvsA(X, Alab="A", Mlab="M", Alim=c(0, 16), Mlim=c(-1, 1) * diff(Alim), pch="
```

Arguments

`X`	Nx2 `matrix` with two channels and N observations.
`Alab,Mlab`	Labels on the x and y axes.
`Alim,Mlim`	Plot range on the A and M axes.
`pch`	Plot symbol used.
`...`	Additional arguments accepted by `points`.
`add`	If `TRUE`, data points are plotted in the current plot, otherwise a new plot is created.

Details

Red channel is assumed to be in column one and green in column two. Log-ratio are calculated taking channel one over channel two.

Value

Returns nothing.

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

plotMvsAPairs.matrix *Plot log-ratios/log-intensities for all unique pairs of data vectors*

Description

Plot log-ratios/log-intensities for all unique pairs of data vectors.

Usage

```
## S3 method for class 'matrix':
plotMvsAPairs(X, Alab="A", Mlab="M", Alim=c(0, 16), Mlim=c(-1, 1) * diff(Alim),
```

Arguments

`X`	NxK `matrix` where N is the number of observations and K is the number of channels.
`Alab,Mlab`	Labels on the x and y axes.
`Alim,Mlim`	Plot range on the A and M axes.
`pch`	Plot symbol used.
`...`	Additional arguments accepted by `points`.
`add`	If `TRUE`, data points are plotted in the current plot, otherwise a new plot is created.

Details

Log-ratios and log-intensities are calculated for each neighboring pair of channels (columns) and plotted. Thus, in total there will be K-1 data set plotted.

The colors used for the plotted pairs are 1, 2, and so on. To change the colors, use a different color palette.

Value

Returns nothing.

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

`plotMvsMPairs.matrix` *Plot log-ratios vs log-ratios for all pairs of columns*

Description

Plot log-ratios vs log-ratios for all pairs of columns.

Usage

```
## S3 method for class 'matrix':
plotMvsMPairs(X, xlab="M", ylab="M", xlim=c(-1, 1) * 6, ylim=xlim, pch=".", ...,
```

Arguments

`X`	Nx2K `matrix` where N is the number of observations and 2K is an even number of channels.
`xlab,ylab`	Labels on the x and y axes.
`xlim,ylim`	Plot range on the x and y axes.
`pch`	Plot symbol used.
`...`	Additional arguments accepted by `points`.
`add`	If `TRUE`, data points are plotted in the current plot, otherwise a new plot is created.

Details

Log-ratio are calculated by over paired columns, e.g. column 1 and 2, column 3 and 4, and so on.

Value

Returns nothing.

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

`plotXYCurve.matrix` *Plot the relationship between two variables as a smooth curve*

Description

Plot the relationship between two variables as a smooth curve.

Usage

```
## S3 method for class 'matrix':
plotXYCurve(X, Y, col=1:nrow(X), lwd=NULL, xlim=NULL, ylim=xlim, xlab=NULL, ylab
```

Arguments

`X, Y`	Two `numeric` NxK `matrix`.
`col`	A `vector` of colors to be used for each of columns.
`lwd`	A `vector` of line widths to be used for each of columns.
`xlim, ylim`	The x and y plotting limits.
`xlab, ylab`	The x and y labels.
`...`	Additional arguments passed to `plotXYCurve.numeric`().
`add`	If `TRUE`, the graph is added to the current plot, otherwise a new plot is created.

Value

Returns (invisibly) the curve fit.

Missing values

Data points (x,y) with non-finite values are excluded.

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

See Also

Internally `plotXYCurve.numeric`() is used.

plotXYCurve.numeric

Plot the relationship between two variables as a smooth curve

Description

Plot the relationship between two variables as a smooth curve.

Usage

```
## S3 method for class 'numeric':
plotXYCurve(x, y, lwd=2, col=1, dlwd=1, dcol=NA, xlim=NULL, ylim=xlim, xlab=NULL
```

Arguments

`x, y`	Two `numeric` `vector` of length N.
`lwd`	The width of the curve.
`col`	The color of the curve.
`dlwd`	The width of the density curves.
`dcol`	The fill color of the interior of the density curves.
`xlim, ylim`	The x and y plotting limits.
`xlab, ylab`	The x and y labels.
`curveFit`	The `function` used to fit the curve. The two first arguments of the function must take `x` and `y`, and the function must return a `list` with fitted elements `x` and `y`.
`...`	Additional arguments passed to `lines` used to draw the curve.
`add`	If `TRUE`, the graph is added to the current plot, otherwise a new plot is created.

Value

Returns (invisibly) the curve fit.

Missing values

Data points (x,y) with non-finite values are excluded.

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

See Also

`plotXYCurve.matrix`().

`robustSmoothSpline` *Robust fit of a Smoothing Spline*

Description

Fits a smoothing spline robustly using the L_1 norm. Currently, the algorithm is an *iterative reweighted smooth spline* algorithm which calls `smooth.spline(x,y,w,...)` at each iteration with the weights `w` equal to the inverse of the absolute value of the residuals for the last iteration step.

Usage

```
## Default S3 method:
robustSmoothSpline(x, y=NULL, w=NULL, ..., minIter=3, maxIter=max(minIter, 50),
```

Arguments

`x`	a `vector` giving the values of the predictor variable, or a `list` or a two-column `matrix` specifying `x` and `y`. If `x` is of class `smooth.spline` then `x$x` is used as the `x` values and `x$yin` are used as the `y` values.
`y`	responses. If `y` is missing, the responses are assumed to be specified by `x`.
`w`	a `vector` of weights the same length as `x` giving the weights to use for each element of `x`. Default value is equal weight to all values.
`...`	Other arguments passed to `smooth.spline`.
`minIter`	the minimum number of iterations used to fit the smoothing spline robustly. Default value is 3.
`maxIter`	the maximum number of iterations used to fit the smoothing spline robustly. Default value is 25.
`sdCriteria`	Convergence criteria, which the difference between the standard deviation of the residuals between two consecutive iteration steps. Default value is 2e-4.
`reps`	Small positive number added to residuals to avoid division by zero when calculating new weights for next iteration.
`plotCurves`	If `TRUE`, the fitted splines are added to the current plot, otherwise not.

Value

Returns an object of class `smooth.spline`.

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

See Also

`smooth.spline`.

Examples

```
data(cars)
attach(cars)
plot(speed, dist, main = "data(cars)  &  robust smoothing splines")

# Fit a smoothing spline using L_2 norm
cars.spl <- smooth.spline(speed, dist)
lines(cars.spl, col = "blue")

# Fit a smoothing spline using L_1 norm
cars.rspl <- robustSmoothSpline(speed, dist)
lines(cars.rspl, col = "red")

# Fit a smoothing spline using L_2 norm with 10 degrees of freedom
lines(smooth.spline(speed, dist, df=10), lty=2, col = "blue")

# Fit a smoothing spline using L_1 norm with 10 degrees of freedom
lines(robustSmoothSpline(speed, dist, df=10), lty=2, col = "red")

legend(5,120, c(
    paste("smooth.spline [C.V.] => df =",round(cars.spl$df,1)),
    paste("robustSmoothSpline [C.V.] => df =",round(cars.rspl$df,1)),
    "standard with s( * , df = 10)", "robust with s( * , df = 10)"
  ), col = c("blue","red","blue","red"), lty = c(1,1,2,2), bg='bisque')
```

---

sampleCorrelations.matrix

Calculates the correlation for random pairs of observations

Description

Calculates the correlation for random pairs of observations.

Usage

```
## S3 method for class 'matrix':
sampleCorrelations(X, MARGIN=1, pairs=NULL, npairs=max(5000, nrow(X)), ...)
```

Arguments

`X`	An NxK `matrix` where N >= 2 and K >= 2.
`MARGIN`	The dimension (1 or 2) in which the observations are. If `MARGIN==1` (`==2`), each row (column) is an observation.
`pairs`	If a Lx2 `matrix`, the L index pairs for which the correlations are calculated. If `NULL`, pairs of observations are sampled.
`npairs`	The number of correlations to calculate.
`...`	Not used.

Value

Returns a `double` `vector` of length `npairs`.

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

References

[1] A. Ploner, L. Miller, P. Hall, J. Bergh & Y. Pawitan. *Correlation test to assess low-level processing of high-density oligonucleotide microarray data*. BMC Bioinformatics, 2005, vol 6.

See Also

`sample`().

Examples

```
# Simulate 20000 genes with 10 observations each
X <- matrix(rnorm(n=20000), ncol=10)

# Calculate the correlation for 5000 random gene pairs
cor <- sampleCorrelations(X, npairs=5000)
print(summary(cor))
```

`sampleTuples` *Sample tuples of elements from a set*

Description

Sample tuples of elements from a set. The elements within a sampled tuple are unique, i.e. no two elements are the same.

Usage

```
## Default S3 method:
sampleTuples(x, size, length, ...)
```

Arguments

`x`	A set of elements to sample from.
`size`	The number of tuples to sample.
`length`	The length of each tuple.
`...`	Additional arguments passed to `sample`().

Value

Returns a NxK `matrix` where N = `size` and K = `length`.

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

See Also

sample().

Examples

```
pairs <- sampleTuples(1:10, size=5, length=2)
print(pairs)

triples <- sampleTuples(1:10, size=5, length=3)
print(triples)

# Allow tuples with repeated elements
quadruples <- sampleTuples(1:3, size=5, length=4, replace=TRUE)
print(quadruples)
```

---

weightedMedian *Weighted Median Value*

Description

Computes a weighted median of a numeric vector.

Usage

```
## Default S3 method:
weightedMedian(x, w, na.rm=NA, interpolate=is.null(ties), ties=NULL, method=c("q
```

Arguments

`x`	a `numeric vector` containing the values whose weighted median is to be computed.
`w`	a vector of weights the same length as `x` giving the weights to use for each element of `x`. Negative weights are treated as zero weights. Default value is equal weight to all values.
`na.rm`	a logical value indicating whether `NA` values in `x` should be stripped before the computation proceeds, or not. If `NA`, no check at all for `NA`s is done. Default value is `NA` (for effiency).
`interpolate`	If `TRUE`, linear interpolation is used to get a consistant estimate of the weighted median.
`ties`	If `interpolate == FALSE`, a character string specifying how to solve ties between two `x`'s that are satisfying the weighted median criteria. Note that at most two values can satisfy the criteria. When `ties` is `"min"`, the smaller value of the two is returned and when it is `"max"`, the larger value is returned. If `ties` is `"mean"`, the mean of the two values is returned and if it is `"both"`, both values are returned. Finally, if `ties` is `"weighted"` (or `NULL`) a weighted average of the two are returned, where the weights are weights of all values `x[i] <= x[k]` and `x[i] >= x[k]`, respectively.
`method`	If `"shell"`, then `order()` is used and when `method="quick"`, then internal `qsort()` is used.
`...`	Not used.

Details

For the `n` elements `x = c(x[1], x[2], ..., x[n])` with positive weights `w = c(w[1], w[2], ..., w[n])` such that `sum(w) = S`, the *weighted median* is defined as the element `x[k]` for which the total weight of all elements `x[i] < x[k]` is less or equal to `S/2` and for which the total weight of all elements `x[i] > x[k]` is less or equal to `S/2` (c.f. [1]).

If `w` is missing then all elements of `x` are given the same positive weight. If all weights are zero, `NA` is returned.

If one or more weights are `Inf`, it is the same as these weights have the same weight and the others has zero. This makes things easier for cases where the weights are result of a division with zero. In this case `median()` is used internally.

When all the weights are the same (after values with weight zero are excluded and `Inf`'s are taken care of), `median` is used internally.

The weighted median solves the following optimization problem:

$$\alpha^* = \arg_\alpha \min \sum_{k=1} K w_k |x_k - \alpha|$$

where $x = (x_1, x_2, \ldots, x_K)$ are scalars and $w = (w_1, w_2, \ldots, w_K)$ are the corresponding "weights" for each individual x value.

Value

Returns the weighted median.

Benchmarks

When implementing this function speed has been highly prioritized and it also making use of the internal quick sort algorithm (from R v1.5.0). The result is that `weightedMedian(x)` is about half as slow as `median(x)`. It is hard to say how much since it depends on the data set, but it is also hard to time it exactly since internal garbage collector etc might mess up the measurements.

Initial test also indicates that `method="shell"`, which uses `order()` is slower than `method="quick"`, which uses internal `qsort()`. Non-weighted median can use partial sorting which is faster because all values do not have to be sorted.

See examples below for some simple benchmarking tests.

Author(s)

Henrik Bengtsson and Ola Hössjer, Centre for Mathematical Sciences, Lund University. Thanks to Roger Koenker, Econometrics, University of Illinois, for the initial ideas.

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, The MIT Press, Massachusetts Institute of Technology, 1989.

See Also

`median`, `mean`() and `weighted.mean`.

Examples

```
x <- 1:10
n <- length(x)

m1 <- median(x)                           # 5.5
m2 <- weightedMedian(x)                   # 5.5
stopifnot(identical(m1, m2))

w <- rep(1, n)
m1 <- weightedMedian(x, w)                # 5.5 (default)
m2 <- weightedMedian(x, ties="weighted")  # 5.5 (default)
m3 <- weightedMedian(x, ties="min")       # 5
m4 <- weightedMedian(x, ties="max")       # 6
stopifnot(identical(m1,m2))

# Pull the median towards zero
w[1] <- 5
m1 <- weightedMedian(x, w)                # 3.5
y <- c(rep(0,w[1]), x[-1])                # Only possible for integer weights
m2 <- median(y)                           # 3.5
stopifnot(identical(m1,m2))

# Put even more weight on the zero
w[1] <- 8.5
weightedMedian(x, w)                # 2

# All weight on the first value
w[1] <- Inf
weightedMedian(x, w)                # 1

# All weight on the last value
w[1] <- 1
w[n] <- Inf
weightedMedian(x, w)                # 10

# All weights set to zero
w <- rep(0, n)
weightedMedian(x, w)                # NA

# Simple benchmarking
bench <- function(N=1e5, K=10) {
  x <- rnorm(N)
  t <- c()
  gc()
  t[1] <- system.time(for (k in 1:K) median(x))[3]
  gc()
  t[2] <- system.time(for (k in 1:K) weightedMedian(x, method="quick"))[3]
  gc()
  t[3] <- system.time(for (k in 1:K) weightedMedian(x, method="shell"))[3]
  t <- t / t[1]
  t[4] <- t[2]/t[3]
  names(t) <- c("median", "wMed-quick", "wMed-shell", "quick/shell")
  t
}

print(bench(N=  5, K=1000))
print(bench(N=100, K=1000))
print(bench(N=1e3, K=100))
print(bench(N=1e5, K=10))
print(bench(N=1e6, K=1))
```

`wpca.matrix` *Light-weight Weighted Principal Component Analysis*

Description

Calculates the (weighted) principal components of a matrix, that is, finds a new coordinate system (not unique) for representing the given multivariate data such that i) all dimensions are orthogonal to each other, and ii) all dimensions have maximal variances.

Usage

```
## S3 method for class 'matrix':
wpca(x, w=NULL, center=TRUE, scale=FALSE, method=c("dgesdd", "dgesvd", "dsvdc"),
```

Arguments

`x`	An NxK `matrix`.
`w`	An N `vector` of weights for each row (observation) in the data matrix. If `NULL`, all observations get the same weight, that is, standard PCA is used.
`center`	If `TRUE`, the (weighted) sample mean column `vector` is subtracted from each column in `mat`, first. If data is not centered, the effect will be that a linear subspace that goes through the origin is fitted.
`scale`	If `TRUE`, each column in `mat` is divided by its (weighted) root-mean-square of the centered column, first.
`method`	If `"dgesdd"` LAPACK's divide-and-conquer based SVD routine is used (faster [1]), if `"dgesvd"`, LAPACK's QR-decomposition-based routine is used, and if `"dsvdc"`, LINPACK's DSVDC(?) routine is used. The latter is just for pure backward compatibility with R v1.7.0.
`swapDirections`	If `TRUE`, the signs of eigenvectors that have more negative than positive components are inverted. The signs of corresponding principal components are also inverted. This is only of interest when for instance visualizing or comparing with other PCA estimates from other methods, because the PCA (SVD) decompostion of a matrix is not unique.
`...`	Not used.

Value

Returns a `list` with elements:

`pc`	An NxK `matrix` where the column `vector`s are the principal components (a.k.a. loading vectors, spectral loadings or factors etc).
`d`	An K `vector` containing the eigenvalues of the principal components.
`vt`	An KxK `matrix` containing the eigenvector of the principal components.
`xMean`	The center coordinate.

It holds that `x == t(t(fit$pc %*% fit$vt) + fit$xMean)`.

Method

A singular value decomposition (SVD) is carried out. Let X=`mat`, then the SVD of the matrix is $X = UDV'$, where U and V are othogonal, and D is a diagonal matrix with singular values. The principal returned by this method are UD.

Internally `La.svd()` (or `svd()`) of the **base** package is used. For a popular and well written introduction to SVD see for instance [2].

Author(s)

Henrik Bengtsson (`http://www.braju.com/R/`)

References

[1] J. Demmel and J. Dongarra, *DOE2000 Progress Report*, 2004. `http://www.cs.berkeley.edu/~demmel/DOE2000/Report0100.html`
[2] Todd Will, *Introduction to the Singular Value Decomposition*, UW-La Crosse, 2004. `http://www.uwlax.edu/faculty/will/svd/`

See Also

For a iterative re-weighted PCA method, see `*iwpca()`. For Singular Value Decomposition, see `svd`(). For other implementations of Principal Component Analysis functions see (if they are installed): `prcomp` in package **stats** and `pca()` in package **pcurve**.

Examples

```
  for (zzz in 0) {

# This example requires plot3d() in R.basic [http://www.braju.com/R/]
if (!require(R.basic)) break

# -------------------------------------------------------------
# A first example
# -------------------------------------------------------------
# Simulate data from the model y <- a + bx + eps(bx)
x <- rexp(1000)
a <- c(2,15,3)
b <- c(2,3,15)
bx <- outer(b,x)
eps <- apply(bx, MARGIN=2, FUN=function(x) rnorm(length(x), mean=0, sd=0.1*x))
y <- a + bx + eps
y <- t(y)

# Add some outliers by permuting the dimensions for 1/3 of the observations
idx <- sample(1:nrow(y), size=1/3*nrow(y))
y[idx,] <- y[idx,c(2,3,1)]

# Down-weight the outliers W times to demonstrate how weights are used
W <- 10

# Plot the data with fitted lines at four different view points
N <- 4
theta <- seq(0,180,length=N)
phi <- rep(30, length.out=N)
# Use a different color for each set of weights
col <- topo.colors(W)

opar <- par(mar=c(1,1,1,1)+0.1)
layout(matrix(1:N, nrow=2, byrow=TRUE))
for (kk in seq(theta)) {
  # Plot the data
  plot3d(y, theta=theta[kk], phi=phi[kk])

  # First, same weights for all observations
  w <- rep(1, length=nrow(y))

  for (ww in 1:W) {
    # Fit a line using IWPCA through data
    fit <- wpca(y, w=w, swapDirections=TRUE)

    # Get the first principal component
    ymid <- fit$xMean
    d0 <- apply(y, MARGIN=2, FUN=min) - ymid
    d1 <- apply(y, MARGIN=2, FUN=max) - ymid
    b <- fit$vt[1,]
    y0 <- -b * max(abs(d0))
    y1 <-  b * max(abs(d1))
    yline <- matrix(c(y0,y1), nrow=length(b), ncol=2)
    yline <- yline + ymid

    points3d(t(ymid), col=col)
    lines3d(t(yline), col=col)

    # Down-weight outliers only, because here we know which they are.
    w[idx] <- w[idx]/2
  }

  # Highlight the last one
  lines3d(t(yline), col="red", lwd=3)
}

par(opar)

} # for (zzz in 0)
rm(zzz)


  if (dev.cur() > 1) dev.off()

  # --------------------------------------------------------------
# A second example
# --------------------------------------------------------------
# Data
x <- c(1,2,3,4,5)
y <- c(2,4,3,3,6)

opar <- par(bty="L")
opalette <- palette(c("blue", "red", "black"))
xlim <- ylim <- c(0,6)
```

```
# Plot the data and the center mass
plot(x,y, pch=16, cex=1.5, xlim=xlim, ylim=ylim)
points(mean(x), mean(y), cex=2, lwd=2, col="blue")


# Linear regression y ~ x
fit <- lm(y ~ x)
abline(fit, lty=1, col=1)

# Linear regression y ~ x through without intercept
fit <- lm(y ~ x - 1)
abline(fit, lty=2, col=1)


# Linear regression x ~ y
fit <- lm(x ~ y)
c <- coefficients(fit)
b <- 1/c[2]
a <- -b*c[1]
abline(a=a, b=b, lty=1, col=2)

# Linear regression x ~ y through without intercept
fit <- lm(x ~ y - 1)
b <- 1/coefficients(fit)
abline(a=0, b=b, lty=2, col=2)


# Orthogonal linear "regression"
fit <- wpca(cbind(x,y))

b <- fit$vt[1,2]/fit$vt[1,1]
a <- fit$xMean[2]-b*fit$xMean[1]
abline(a=a, b=b, lwd=2, col=3)

# Orthogonal linear "regression" without intercept
fit <- wpca(cbind(x,y), center=FALSE)
b <- fit$vt[1,2]/fit$vt[1,1]
a <- fit$xMean[2]-b*fit$xMean[1]
abline(a=a, b=b, lty=2, lwd=2, col=3)

legend(xlim[1],ylim[2], legend=c("lm(y~x)", "lm(y~x-1)", "lm(x~y)",
          "lm(x~y-1)", "pca", "pca w/o intercept"), lty=rep(1:2,3),
                     lwd=rep(c(1,1,2),each=2), col=rep(1:3,each=2))

palette(opalette)
par(opar)
```

Index