The Channel class

Description

An object of the Channel class represents a single kind of measurement performed at all spots of a microarray channel. These objects are essentially just vectors of data, with length equal to the number of spots on the microarray, with some extra metadata attached.

Usage

Channel(parent, name, type, x)
## S4 method for signature 'Channel, missing':
plot(object, ...)
## S4 method for signature 'Channel':
hist(object, ...)
## S4 method for signature 'Channel':
summary(object, ...)
## S4 method for signature 'Channel':
print(object, ...)
## S4 method for signature 'Channel':
image(object, ...)

Arguments

parent    A string representing the name of a parent object from which this object was derived
name      A string with a displayable name for this object
type      A ChannelType object
x         A numeric vector
object    A Channel object
...       Additional arguments are as in the underlying generic methods.

Details

As described in the help pages for ChannelType, each microarray hybridization experiment produces one or more channels of data. Channel objects represent a single measurement performed at spots in one microarray channel. The raw data from a full experiment typically contains multiple measurements in multiple channels.

The full set of measurements is often highly processed (by, for example, background subtraction, normalization, log transformation, etc.) before it becomes useful. We have added a history slot that keeps track of how a Channel was produced. By allowing each object to maintain a record of its history, it becomes easier to document the processing when writing up the methods for reports or papers. The history slot of the object is updated using the generic function process together with a Processor object.
Value

The print, hist, and image methods all invisibly return the Channel object on which they were invoked. The print and summary methods return nothing.

Slots

parent: A string representing the name of a parent object from which this object was derived.

name: A string with a displayable name for this object

type: A ChannelType object

x: A numeric vector

history: A list that keeps a record of the calls used to produce this object

Methods

print(object, ...) Print all the data on the object. Since this includes the entire data vector, you rarely want to do this.

summary(object, ...) Write out a summary of the object.

plot(object, ...) Produce a scatter plot of the measurement values in the slot x of the object against their index, which serves as a surrogate for the position on the microarray. Additional graphical parameters are passed along.

hist(object, ...) Produce a histogram of the data values in slot x of the object. Additional graphical parameters are passed along.

image(object, ...) This method produces a two-dimensional "cartoon" image of the measurement values, with the position in the cartoon corresponding to the two-dimensional arrangement of spots on the actual microarray. Additional graphical parameters are passed along.

Author(s)

Kevin R. Coombes <kcoombes@mdanderson.org>

See Also

ChannelType, process, Processor

Examples

# simulate a moderately realistic looking microarray
nc <- 100          # number of rows
nr <- 100          # number of columns
v <- rexp(nc*nr, 1/1000)  # "true" signal intensity (vol)
b <- rnorm(nc*nr, 80, 10)  # background noise
s <- sapply(v-b, max, 1)  # corrected signal intensity (svol)
ct <- ChannelType('user', 'random', nc, nr, 'fake')
raw <- Channel(name='fraud', type=ct, parent='', x=v)
subbed <- Channel(name='fraud', parent='', type=ct, x=s)
rm(nc, nr, v, b, s)  # clean some stuff
summary(subbed)
summary(raw)

par(mfrow=c(2,1))
plot(raw)
hist(raw)

par(mfrow=c(1,1))
image(raw)

# finish the cleanup
rm(ct, raw, subbed)

---

**ChannelType-class**  
*The ChannelType class*

---

**Description**

This class represents the "type" of a microarray channel.

**Usage**

```r
ChannelType(mk, md, nc, nr, gl, design = "")
setDesign(object, design)
getDesign(object)
## S4 method for signature 'ChannelType':
print(x,...)
## S4 method for signature 'ChannelType':
summary(object,...)
```

**Arguments**

- `mk`  
  A string; the name of the manufacturer of the microarray (e.g., 'Affymetrix')
- `md`  
  A string; the model of the microarray (e.g., 'Hu95A')
- `nc`  
  An integer; the number of columns in the array
- `nr`  
  An integer; the number of rows in the array
- `gl`  
  A string; the material used to label samples
- `design`  
  A string containing the name of an object describing details about the design of the microarray
- `object`  
  A ChannelType object
- `x`  
  A ChannelType object
- `...`  
  Additional arguments are as in the underlying generic methods.
Details

Microarrays come in numerous flavors. At present, the two most common types are the synthesized oligonucleotide arrays produced by Affymetrix and the printed cDNA arrays on glass, which started in Pat Brown’s lab at Stanford. In earlier days, it was also common to find nylon microarrays, with the samples labeled using a radioactive isotope. The glass arrays are distinguished from other kinds of arrays in that they typically cohybridize two different samples simultaneously, using two different fluorescent dyes. The fluorescence from each dye is scanned separately, producing two images and thus two related sets of data from the same microarray. We refer to these parallel data sets within an array as “channels”.

An object of the ChannelType class represents a combination of the kind of microarray along with the kind of labeling procedure. These objects are intended to be passed around as part of more complex objects representing the actual gene expression data collected from particular experiments, in order to be able to eventually tie back into the description of what spots were laid down when the array was produced.

The ChannelType object only contains a high level description of the microarray, however. Detailed information about what biological material was laid down at each spot on the microarray is stored elsewhere, in a ‘design’ object. Within a ChannelType object, the design is represented simply by a character string. This string should be the name of a separate object containing the detailed design information. This implementation allows us to defer the design details until later. It also saves space by putting the details in a single object instead of copying them into every microarray. Finally, it allows that single object to be updated when better biological annotations are available, with the benefits spreading immediately to all the microarray projects that use that design.

Value

The ChannelType constructor reutrns a valid object of the class.

The setDesign function invisibly returns the ChannelType object on which it was invoked.

The getDesign function returns the design object referred to by the design slot in the ChannelType object. If this string does not evaluate to the name of an object, then getDesign returns a NULL value.

Slots

    maker: A string; the name of the manufacturer of the microarray (e.g., 'Affymetrix')
    model: A string; the model of the microarray (e.g., 'Hu95A')
    nCol: An integer; the number of columns in the array
    nRow: An integer; the number of rows in the array
    glow: A string; the material used to label samples
    design: A string containing the name of an object describing details about the design of the microarray

Methods

    print(x, ...) Prints all the information in the object
    summary(object, ...) Writes out a summary of the object

4
Author(s)

Kevin R. Coombes <kcoombes@mdanderson.org>

See Also

Channel

Examples

```r
x <- ChannelType('Affymetrix', 'oligo', 100, 100, 'fluor')
print(x)

summary(x)

y <- setDesign(x, 'fake.design')
print(y)

summary(y)
d <- getDesign(y)
d

rm(d, x, y) # cleanup
```

---

ColorCodedPair-class  The ColorCodedPair class

Description

This class represents a vector of (x,y) pairs, each of which should be plotted in a specific color with a specific symbol.

Usage

ColorCodedPair(x, y, ccl)
## S4 method for signature 'ColorCodedPair, missing':
plot(object, ...)

Arguments

- **x**: A numeric vector
- **y**: A numeric vector
- **ccl**: A list of ColorCoding objects
- **object**: A ColorCodedPair object
- **...**: Additional arguments are as in the underlying generic methods.
Details

It is often necessary with microarray data to produce multiple plots, where each point on the plot corresponds to a gene or a spot on the microarray. Across multiple plots, we often want to use symbols or colors to mark subsets of genes with certain properties. The ColorCodedPair class works in tandem with the ColorCoding class to make it easier to maintain consistent plotting conventions across multiple graphs.

Value

The constructor returns a valid ColorCodedPair object.

The plot method invisibly returns the object being plotted.

Slots

x A numeric vector
y A numeric vector
colorCodingList A list of ColorCoding objects

Methods

plot(object, ...) Plot the ColorCodedPair object, with appropriate colors and symbols (of course).

Author(s)

Kevin R. Coombes <kcoombes@mdanderson.org>

See Also

The ColorCoding class, colorCode

Examples

theta <- (0:360)*pi/180
x <- cos(theta)
y <- sin(theta)
xp <- x > 0
yp <- y > 0
colors <- list(ColorCoding(xp&yp, COLOR.EXPECTED),
               ColorCoding(xp&!yp, COLOR.OBSERVED),
               ColorCoding(!xp&yp, COLOR.PERMTEST),
               ColorCoding(!xp&!yp, COLOR.FITTED))
plot(ColorCodedPair(x, y, colors))
plot(ColorCodedPair(theta, x, colors))
plot(ColorCodedPair(theta, y, colors),
     xlab='angle in radians', ylab='sine', main='colored sine')
The ColorCoding class

Description

A class for associating plotting symbols and colors with a logical vector or with levels of a factor.

Usage

ColorCoding(v, color, mark = 1)
colorCode(fac, colorScheme = 1:length(levels(fac)), mark = 1)

Arguments

v                      a logical vector
color                  a string or integer representing a color
mark                   an integer representing a plotting symbol, or list of plotting symbols that
                        should be associated with levels of the factor fac
fac                    A factor
colorScheme            A list of colors that should be associated with levels of the factor fac

Details

It is often necessary with microarray data to produce multiple plots, where each point on
the plot corresponds to a gene or a spot on the microarray. Across multiple plots, we often
want to use symbols or colors to mark subsets of genes with certain properties. The
ColorCoding class works in tandem with the ColorCodedPair class to make it easier to
maintain consistent plotting conventions across multiple graphs.

Value

The constructor returns a valid object of the ColorCoding class.

The colorCode function provides a simple interface to associate colors and symbols with
the levels of a factor. It returns a list of valid ColorCoding objects that can be passed
directly to the constructor of a ColorCodedPair object.

Creating Objects

Although objects can be created using new, the preferred method is to use the constructor
function, ColorCoding. To create a set of colors and symbols associated with different
levels of a factor, use the colorCode function.
Slots

  v  a logical vector
  color  a string or integer representing a color
  mark  an integer representing a plotting symbol

Methods

  There are no specialized methods for objects of this class; all activities can be performed by directly accessing the documented slots.

Author(s)

  Kevin R. Coombes <kcoombes@mdanderson.org>

See Also

  The ColorCodedPair class

Examples

theta <- (0:360)*pi/180
x <- cos(theta)
y <- sin(theta)
xp <- x > 0
yp <- y > 0
colors <- list(ColorCoding(xp&yp, COLOR.BORING),
               ColorCoding(xp&!yp, COLOR.TOP.TEN),
               ColorCoding(!xp&yp, COLOR.BOTTOM.TEN),
               ColorCoding(!xp&!yp, COLOR.CONFIDENCE.CURVE))
plot(ColorCodedPair(x, y, colors))

plot(ColorCodedPair(theta, x, colors))

plot(ColorCodedPair(theta, y, colors),
     xlab='angle in radians', ylab='sine', main='colored sine')

fac <- factor(rep(c('left', 'right'), c(180, 181)))
colors <- colorCode(fac, c('blue', 'red'))
plot(ColorCodedPair(x, y, colors))

# cleanup
rm(x, y, xp, yp, theta, colors, fac)

colorList <- c(COLOR.BORING, COLOR.SIGNIFICANT,
               COLOR.EXPECTED, COLOR.OBSERVED,
               COLOR.PERMTEST, COLOR.FITTED,
               COLOR.CENTRAL.LINE, COLOR.CONFIDENCE.CURVE,
               COLOR.BAD.REPLICATE, COLOR.WORST.REPLICATE,
COLOR.FOLD.DIFFERENCE, COLOR.BAD.REPLICATE.RATIO,
COLOR.TOP.TEN, COLOR.BOTTOM.TEN,
COLOR.TOP.TEN.SOLO, COLOR.BOTTOM.TEN.SOLO
)
plot(c(1,4), c(1,4), type='n')
for (i in 1:4) {
  for (j in 1:4) {
    points(i,j, col=colorList[i + 4*(j-1)], pch=16, cex=4)
  }
}
rm(colorList, i, j)

CompleteChannel-class

The CompleteChannel class

Description
An object of the CompleteChannel class represents one channel (red or green) of a two-color fluorescence microarray experiment. Alternatively, it can also represent the entirety of a radioactive microarray experiment. Affymetrix experiments produce data with a somewhat different structure because they use multiple probes for each target gene.

Usage

CompleteChannel(name, type, data)

## S4 method for signature 'CompleteChannel':
print(x, ...)
## S4 method for signature 'CompleteChannel':
summary(object, ...)
## S4 method for signature 'CompleteChannel':
as.data.frame(x, row.names=NULL, optional=FALSE)
## S4 method for signature 'CompleteChannel, missing':
plot(x, useLog=FALSE, ...)
## S4 method for signature 'CompleteChannel':
image(x, ...)
## S4 method for signature 'CompleteChannel':
analyze(object, useLog=FALSE, ...)
## S4 method for signature 'CompleteChannel, Processor':
process(object, action, parameter)
## S4 method for signature 'CompleteChannel':
channelize(object)

Arguments

name A string containing the name of the object
type A ChannelType object
data  A data frame. For the pre-defined “extraction” processors to work correctly, this should include columns called vol, bkgd, svol, SD, and SN.

x  A CompleteChannel object

object  A CompleteChannel object

useLog  A logical value

action  A Processor object used to process a CompleteChannel.

parameter  Any object that makes sense as a parameter to the function represented by the Processor action

row.names  See as.data.frame

optional  See as.data.frame

...  Additional arguments are as in the underlying generic methods.

Details

The names come from the default column names in the ArrayVision software package used at M.D. Anderson for quantifying glass or nylon microarrays. Column names used by other software packages should be mapped to these.

Value

The analyze method returns a list of three density functions.

The return value of the process function depends on the Processor performing the action, but is typically a Channel object.

Graphical methods invisibly return the object on which they were invoked.

Slots

name: A string containing the name of the object

type: A ChannelType object

data: A data frame

history: A list that keeps a record of the calls used to produce this object

Methods

print(x, ...) Print all the data on the object. Since this includes the data frame, you rarely want to do this.

summary(object, ...) Write out a summary of the object.

as.data.frame(x,row.names=NULL, optional=FALSE) Convert the CompleteChannel object into a data frame. As you might expect, this simply returns the data frame in the data slot of the object.

plot(x, useLog=FALSE, ...) Produces three estimated density plots: one for the signal, one for the background, and one for the background-corrected signal. Additional graphical parameters are passed along. The logical flag useLog determines whether the data are log-transformed before estimating and plotting densities.
analyze(object, useLog=FALSE, ...) This method computes the estimated probability density functions for the three data components (signal, background, and background-corrected signal), and returns them as a list.

image(object, ...) Uses the image method for Channel objects to produce geographically aligned images of the log-transformed intensity and background estimates.

cannelize(object) A string giving the name of the class of a channel that is produced when you process a CompleteChannel object.

process(object, action, parameter=NULL) Use the Processor action to process the CompleteChannel object. Returns an object of the class described by channelize, which defaults to Channel.

Pre-defined Processors

The library comes with several Processor objects already defined; each one takes a CompleteChannel as input, extracts a single value per spot, and produces a Channel as output.

PROC.BACKGROUND Extract the vector of local background measurements.

PROC.SIGAAL Extract the vector of foreground signal intensity measurements.

PROC.CORRECTED.SIGAAL Extract the vector of background-corrected signal measurements.

PROC.NEG.CORRECTED.SIGAAL Extract the vector of background-corrected signal intensities by subtracting the local background from the observed foreground, without truncation.

PROC.SD.SIGAAL Extract the vector of pixel standard deviations of the signal intensity.

PROC.SIGAAL.TO.NOISE Extract the vector of signal-to-noise ratios, defined as CORRECTED.SIGAAL divided by the standard deviation of the background pixels.

Author(s)

Kevin R. Coombes <kcoombes@mdanderson.org>

See Also

process, Processor, Pipeline, Channel, as.data.frame

Examples

# simulate a complete channel object

v <- rexp(10000, 1/1000)
b <- rnorm(10000, 60, 6)
s <- sapply(v-b, function(x) {max(0, x)})
ct <- ChannelType('user', 'random', 100, 100, 'fake')
x <- CompleteChannel(name='fraud', type=ct,
    data=data.frame(vol=v, bkgd=b, svol=s))
rm(v, b, s, ct)

summary(x)
opar <- par(mfrow=c(2,3))
plot(x)
plot(x, main='Log Scale', useLog=TRUE)
par(opar)

opar <- par(mfrow=c(2,1))
image(x)
par(opar)

b <- process(x, PROC.NEG.CORRECTED.SIGNAL)
summary(b)

q <- process(b, PIPELINE.STANDARD)
summary(q)

q <- process(x, PIPELINE.MDACC.DEFAULT)
summary(q)

# cleanup
rm(x, b, q, opar)

---

**Pipeline-class**

*The Pipeline class*

**Description**

A Pipeline represents a standard multi-step procedure for processing microarray data. A Pipeline represents a series of Processors that should be applied in order. You can think of a pipeline as a completely defined (and reusable) set of transformations that is applied uniformly to every microarray in a data set.

**Usage**

```r
## S4 method for signature 'ANY, Pipeline':
process(object, action, parameter=NULL)
## S4 method for signature 'Pipeline':
summary(object, ...)
makeDefaultPipeline(ef = PROC.SIGNAL, ep = 0,
                  nf = PROC.GLOBAL.NORMALIZATION, np = 0,
                  tf = PROC.THRESHOLD, tp = 25,
                  lf = PROC.LOG.TRANSFORM, lp = 2,
                  name = "standard pipe",
                  description = "my method")
```

**Arguments**

- **object**
  In the process method, any object appropriate for the input to the Pipeline. In the summary method, a Pipeline object.

- **action**
  A Pipeline object used to process an object.
parameter: Irrelevant, since the Pipeline ignores the parameter when process is invoked.

... Additional arguments are as in the underlying generic methods.

ef: “Extractor function”: First Processor in the Pipeline, typically a method that extracts a single kind of raw measurement from a microarray.

ep: Default parameter value for ef.

nf: “Normalization function”: Second Processor in the Pipeline, typically a normalization step.

np: Default parameter value for nf.

tf: “Threshold function”: Third Processor in the Pipeline, typically a step that truncates data below at some threshold.

tp: Default parameter value for tf.

lf: “Log function”: Fourth Processor in the Pipeline, typically a log transformation.

lp: Default parameter value for lf.

name: A string; the name of the pipeline.

description: A string; a longer description of the pipeline.

Details

A key feature of a Pipeline is that it is supposed to represent a standard algorithm that is applied to all objects when processing a microarray data set. For that reason, the parameter that can be passed to the process function is ignored, ensuring that the same parameter values are used to process all objects. By contrast, each Processor that is inserted into a Pipeline allows the user to supply a parameter that overrides its default value.

We provide a single constructor, makeDefaultPipeline to build a specialized kind of Pipeline, tailored to the analysis of fluorescently labeled single channels in a microarray experiment. More general Pipelines can be constructed using new.

Value

The return value of the generic function process is always an object related to its input, which keeps a record of its history. The precise class of the result depends on the functions used to create the Pipeline.

Slots

proclist: A list of Processor objects.

name: A string containing the name of the object.

description: A string containing a longer description of the object.

Methods

process(object, action, parameter) Apply the series of functions represented by the Pipeline action to the object, updating its history appropriately. The parameter is ignored, since the Pipeline always uses its default values.

summary(object, ...) Write out a summary of the object.
Pre-defined Pipelines

The library comes with two Pipeline objects already defined

PIEPLINE.STANDARD Takes a Channel object as input. Performs global normalization by rescaling the 75th percentile to 1000, truncates below at 25, then performs log (base-two) transformation.

PIEPLINE.MDACC.DEFAULT Takes a CompleteChannel as input, extracts the raw signal intensity, and then performs the same processing as PIEPLINE.STANDARD.

Author(s)

Kevin R. Coombes <kcoombes@mdanderson.org>

See Also

Channel, process, CompleteChannel

Examples

# simulate a moderately realistic looking microarray
nc <- 100
nr <- 100
v <- rexp(nc*nr, 1/1000)
b <- rnorm(nc*nr, 80, 10)
s <- sapply(v-b, max, 1)
ct <- ChannelType(’user’, ’random’, nc, nr, ’fake’)
subbed <- Channel(name=’fraud’, parent=’’, type=ct, x=s)
rm(ct, nc, nr, v, b, s) # clean some stuff

# example of standard data processing
processed <- process(subbed, PIEPLINE.STANDARD)

summary(processed)

par(mfrow=c(2,1))
plot(processed)
hist(processed)

par(mfrow=c(1,1))
image(processed)

rm(subbed, processed)

Processor-class

The Processor class

Description

A Processor represents a function that acts on the data of a some object to process it in some way. The result is always another related object, which should record some history about exactly how it was processed.
Usage

```r
## S4 method for signature 'Channel, Processor':
process(object, action, parameter=NULL)
## S4 method for signature 'Processor':
summary(object, ...)
```

Arguments

- `object`: In the `process` method, a `Channel` object. In the `summary` method, a `Processor` object.
- `action`: A `Processor` object used to process a `Channel`.
- `parameter`: Any object that makes sense as a parameter to the function represented by the `Processor` action.
- `...`: Additional arguments are as in the underlying generic methods.

Value

The return value of the generic function `process` is always an object related to its `Channel` input, which keeps a record of its history. The precise class of the result depends on the function used to create the `Processor`.

Slots

- `f`: A function that will be used to process microarray-related object
- `default`: The default value of the parameters to the function `f`
- `name`: A string containing the name of the object
- `description`: A string containing a longer description of the object

Methods

- `process(object, action, parameter)`: Apply the function represented by `action` to the `Channel` object, updating the history appropriately. If the `parameter` is `NULL`, then use the default value.
- `summary(object, ...)`: Write out a summary of the object.

Pre-defined Processors

The library comes with several `Processor` objects already defined; each one takes a `Channel` as input and produces a modified `Channel` as output.

- **PROC.SUBTRACTOR**: Subtracts a global constant (default: 0) from the data vector in the `Channel`.
- **PROC.THRESHOLD**: Truncates the data vector below, replacing the values below a threshold (default: 0) with the threshold value.
- **PROC.GLOBAL.NORMALIZATION**: Normalizes the data vector in the `Channel` by dividing by a global constant. If the parameter takes on its default value of 0, then divide by the 75th percentile.
PROC.LOG.TRANSFORM Performs a log transformation of the data vector. The parameter specifies the base of the logarithm (default: 2).

PROC.MEDIAN.EXpressed.NORMALIZATION Normalizes the data vector by dividing by the median of the expressed genes, where “expressed” is taken to mean “greater than zero”.

PROC.SUBSET.NORMALIZATION Normalizes the data vector by dividing by the median of a subset of genes. When the parameter has a default value of 0, then this method uses the global median. Otherwise, the parameter should be set to a logical or numerical vector that selects the subset of genes to be used for normalization.

PROC.SUBSET.MEAN.NORMALIZATION Normalizes the data vector by dividing by the mean of a subset of genes. When the parameter has a default value of 0, then this method uses the global mean. Otherwise, the parameter should be set to a logical or numerical vector that selects the subset of genes to be used for normalization.

Author(s)
Kevin R. Coombes <kcoombes@mdanderson.org>

See Also
Channel, process, Pipeline, CompleteChannel

Examples

# simulate a moderately realistic looking microarray
nc <- 100
nr <- 100
v <- rexp(nc*nr, 1/1000)
b <- rnorm(nc*nr, 80, 10)
s <- sapply(v-b, max, 1)
c <- ChannelType('user', 'random', nc, nr, 'fake')
subbed <- Channel(name='fraud', parent='', type=c, x=s)
rm(c, nc, nr, v, b, s)  # clean some stuff

# example of standard data processing
nor <- process(subbed, PROC.GLOBAL.NORMALIZATION)
thr <- process(nor, PROC.THRESHOLD, 25)
processed <- process(thr, PROC.LOG.TRANSFORM, 2)

summary(processed)

par(mfrow=c(2,1))
plot(processed)
hist(processed)

par(mfrow=c(1,1))
image(processed)

rm(nor, thr, subbed, processed)
channelize

*Object-oriented conversion from CompleteChannel to Channel*

**Description**

Provide a generic function for propagating the class of derived objects through a processing pipeline.

**Usage**

channelize(object)

**Arguments**

object An object of a class derived from CompleteChannel.

**Details**

Having abstracted away the notion of extracting a particular measurement from a CompleteChannel object and producing a simple Channel, we need a way to allow object-oriented programming and derived classes to work with our Processor and Pipeline routines. The underlying idea is that specific kinds of microarrays or specific software to quantify microarrays might have special properties that should be exploited in processing. For example, the first few generations of miroarrays printed at M.D. Anderson spotted every cDNA clone in duplicate. The analysis of such arrays should exploit this additional structure. In order to do so, we must derive classes from CompleteChannel and Channel and ensure that the classes of extracted objects are propagated correctly through the processing pipeline. The channelize methods achieves this goal.

**Value**

Returns a string, which represents the name of a class (suitable for passing to the new constructor) extracted from an object belonging to a class derived from CompleteChannel.

**Author(s)**

Kevin R. Coombes <kcoombes@mdanderson.org>

**See Also**

Processor, Pipeline, Channel, CompleteChannel

**Examples**
Description

New generic functions for processing and analyzing microarrays.

Usage

process(object, action, parameter = NULL)
analyze(object, ...)

Arguments

object Any OOMPA class representing a microarrays or a set of microarrays
action The action to process the class.
parameter Any parameters needed to execute the process.
... Place holder for additional parameters needed in derived classes

Details

In general, the `analyze` method represents an expensive computational step carried out in preparation for a graphical display, but the semantics may differ from class to class. The default implementation of the method performs the null analysis; that is, the return value is identical to the object that is passed in as the first argument.

The `process` method represents a function that acts on the data of some object to process it in some way. For example, normalizing a set of microarray data is typically one processing step in a long series that is required to take the raw data and turn it into something useful.

Value

Varies depending on the implementation in derived classes. Typically another object of the same or a closely related class.

Author(s)

Kevin R. Coombes <kcoombes@mdanderson.org>

See Also

Processor, Pipeline

Examples

# these are generic functions...
graph.utility

OOMPA graphical utility functions

Description

Utility functions for graphics.

Usage

\[ \text{ellipse}(a, b, x0=0, y0=0, \ldots) \]
\[ \text{f.qq}(x, \text{main} = \"\", \text{cut} = 0, \ldots) \]
\[ \text{f.qt}(x, df, \text{main} = \"\", \text{cut} = 0, \ldots) \]

Arguments

- \( a \): Half the length of the elliptical axis in the x-direction
- \( b \): Half the length of the elliptical axis in the y-direction
- \( x0 \): X-coordinate of the center of the ellipse
- \( y0 \): Y-coordinate of the center of the ellipse
- \( \text{main} \): A text string
- \( \text{cut} \): A real number
- \( df \): An integer; the number of degrees of freedom in the t-test
- \( \ldots \): Additional graphical parameters passed on to lower-level functions
- \( x \): A numeric vector

Details

The \text{ellipse} function draws an ellipse on an existing plots. The ellipses produced by this function are oriented with their major and minor axes parallel to the coordinate axes. The current implementation uses \text{points} internally.

The function \text{f.qq} is a wrapper that combines \text{qqnorm} and \text{qqline} into a single function call.

The function \text{f.qt} is a wrapper that produces quantile-quantile plots comparing the observed vector \( x \) with a T-distribution.

Value

Author(s)

Kevin R. Coombes &lt;kcoombes@mdanderson.org&gt;

See Also

See also \text{points}
Examples

```r
x <- rnorm(1000, 1, 2)
y <- rnorm(1000, 1, 2)
plot(x, y)
ellipse(1, 1, col=6, type='l', lwd=2)
ellipse(3, 2, col=6, type='l', lwd=2)
f.qq(x, main='Demo', col='blue')
f.qq(x, cut = 3)
f qt(x, df = 3)
f qt(x, df = 40)
```

---

**matrix.utility**  
*OOMPA Matrix Utility Functions*

**Description**

Utility functions for manipulating matrices.

**Usage**

```r
flipud(x)
fliplr(x)
```

**Arguments**

- **x**  
a matrix

**Value**

The `flipud` function returns a matrix the same size as `x`, with the order of the rows reversed, so the matrix has been flipped vertically. The `fliplr` function returns a matrix the same size as `x` but flipped horizontally, with the order of the columns reversed.

**Author(s)**

Kevin R. Coombes <kcoombes@mdanderson.org>

**Examples**

```r
mat <- matrix(1:6, 2, 3)
mat
flipud(mat)
fliplr(mat)
```
**Description**

A collection of predefined color names to help ensure consistency in multiple graphical displays of microarray data.

**COLOR.BORING**: Used to mark uninteresting points in a plot; gray.

**COLOR.SIGNIFICANT**: Used to mark points that are statistically significant; red

**COLOR.EXPECTED**: Used to draw curves representing an expected distribution; blue

**COLOR.OBSERVED**: Used to draw curves indicating the observed distribution; darkgreen

**COLOR.PERMTEST**: Used to draw curves indicating distributions derived from a permutation test; magenta

**COLOR.FITTED**: Used to draw curves obtained by some fitting procedure, such as loess; orange

**COLOR.CENTRAL.LINE**: Used to draw lines through the centers of distributions or expected values; blue

**COLOR.CONFIDENCE.CURVE**: Used to draw confidence bounds around curves; red3

**COLOR.BAD.REPLICATE**: Used to indicate highly variable points; purple1

**COLOR.WORST.REPLICATE**: Used to mark extraordinarily variable points; purple3

**COLOR.FOLD.DIFFERENCE**: Used to indicate points with large fold difference; skyblue

**COLOR.BAD.REPLICATE.RATIO**: Used to flag points for which the ratios at replicate spots are highly variable; violetred

**COLOR.TOP.TEN**: Used to mark points in the "top ten" list; cadetblue

**COLOR.BOTTOM.TEN**: Used to mark points in "bottom ten" list of most underexpressed genes; pink

**COLOR.BOTTOM.TEN.SOLO**: Use unknown; palegreen

**COLOR.TOP.TEN.SOLO**: Use unknown; deeppink

**Examples**

```r
x <- seq(0, 2*pi, by=0.1)
plot(x, sin(x), col=COLOR.BORING)
```
stat.utility  OOMPA Statistical Utility Functions

Description
Utility functions for statistical computations.

Usage
f.above.thresh(a, t)
f.cord(x, y, inf.rm)
f.oneway.rankings(r, s)
stdize(x, center=TRUE, scale=TRUE)

Arguments
a       a vector
a real number
x       a vector
y       a vector
inf.rm  a logical value
r       s
center  scale

Value
f.above.thresh returns the fraction of elements in the vector a that are greater than the threshold t.
f.cord returns the concordance coefficient between the two input vectors x and y. If inf.rm is true, then infinite values are removed before computing the concordance; missing values are always removed.
stdize is a synonym for scale.
f.oneway.rankings is implemented as order(s)[r] and I cannot recall why we defined it or where we used it.

Author(s)
Kevin R. Coombes <kcoombes@mdanderson.org>
Examples

```r
x <- rnorm(1000, 1, 2)
y <- rnorm(1000, 1, 2)
f.above.thresh(x, 0)
f.above.thresh(y, 0)
f.cord(x, y)
z <- stdize(x)
f.cord(x, z)
```