Package ‘GENEAread’

June 19, 2012

Type Package

Title Package For Reading Binary files

Version 1.0

Date 19/06/2012

Author Zhou Fang <`zhou@activinsights.co.uk`>

Maintainer ActivInsights Ltd. <`joss.langford@activinsights.co.uk`>

Description Functions and analytics for GENEA-compatible accelerometer data into R objects. See topic ’GENEAread’ for an introduction to the package.

License GPL

LazyLoad yes

ByteCompile yes

Depends bitops

Suggests mmap, MASS

R topics documented:

`GENEAread-package` *GENEAread: a package to process binary accelerometer output files.*

Description

This is a package to process binary output files from the GENEA accelerometer data. The main functions are:

read.bin
stft
epoch.apply

Details

Package:	GENEAread
Type:	Package
Version:	1.0
Date:	19/06/2012
License:	GPL
LazyLoad:	yes

Main tasks performed

The main tasks performed by the package are listed below. The relevant topic contains documentation and examples for each.

Extraction of file header material is accomplished by `header.info`.

Input and downsampling of data is accomplished by `read.bin`.

Selection of time intervals is accomplished via `get.intervals`.

Computation of epochal summaries is accomplished by `epoch.apply` and other functions documented therein.

Computation of STFT analyses is accomplished by `stft`.

Classes implemented

The package provides definitions and methods for the following S3 classes:

GRtime: Provides numeric storage and streamlined plotting for times. `GRtime`

AccData: Stores GENEA accelerometer data, allowing plotting, subsetting and other computation. `AccData`

VirtAccData: A virtual AccData object, for just-in-time data access via `get.intervals`.

stft: Processed STFT outputs, for plotting via `plot.stft`.

Author(s)

Zhou Fang <zhou@activinsights.co.uk> ActivInsights Ltd. <joss.langford@activinsights.co.uk>

`AccData` *Methods for processing and summarising AccData.*

Description

A variety of functions and methods for handling processed AccData.

Usage

```
## S3 method for class 'AccData'
x[i=1:dim(x$data.out)[1], j=NULL, drop=TRUE]
## S3 method for class 'AccData'
x$name
## S3 method for class 'AccData'
print(x, ...)
## S3 method for class 'AccData'
summary(object, ...)
## S3 method for class 'AccData'
plot(x, y=NULL, what = c("sd", "mean", "temperature", "light", "voltage"),draw = TRUE, resolutic
```

Arguments

`x, object`	"AccData" object to process.
`i,j`	Coordinates for matrix like manipulation.
`drop`	logical. Coerce to vector if one dimensional?
`name`	list field to extract.
`y`	Optional variable to plot as y-axis.
`what`	Type of plot to create.
`draw`	logical. Whether to plot output.
`resolution`	Approximate number of time steps to plot.
`...`	Additional arguments to pass to default methods.

Details

These functions allow access and manipulation of AccData class objects.

`[` allows matrix style manipulations. If the first column (the timestamp column) is included, a data.frame is produced with the timestamp as a "GRtime" object via `convert.time`. This allows improved plotting of time axes. If j is not specified, an "AccData" object is returned.

`$` allow list style manipulations. In addition to the internal components of "AccData" objects (see `read.bin`), a number of keywords are recognised:

"time": Timestamp "x","y","z": x, y and z accelerometer components "xyz": The three accelerometer components together "temperature" "button" "voltage" "light" "svm": Sum of vector magnitudes

`print` and `summary` both provide useful summaries of the data. Summary returns invisibly an object representation of its output in list format - in particular, it gives summary statistics of epochal standard deviations on a 10 second epoch.

Finally, plot provides a range of useful summary plots, depending on the specification of `what`. To reduce computational requirements, epochs and so on are chosen or downsampling done so that a maximum of around 100 time points are plotted. If `plot` is called with a specified `y`, `x` is considered as its vector of timestamps. If `draw = FALSE`, plot produces no side effects but instead returns the object that would have been plotted.

See Also

`header.info`, `epoch.apply`, `get.intervals`

Examples

```
binfile  = system.file("binfile/TESTfile.bin", package = "GENEAread")[1]

#Read in the entire file, calibrated
procfile<-read.bin(binfile)

print(procfile)
summary(procfile)

plot(procfile$temperature)
plot(procfile[,c(1,7)])
```

`epoch.apply` *Compute epochal summary statistics.*

Description

Computes epochal summary statistics for an "AccData" object, matrix, or vector, and collates into a matrix or vector.

Usage

```
epoch.apply(obj, epoch.size=10, incl.date = FALSE, FUN)

epoch.mean(obj, epoch.size=10, incl.date = FALSE, sqrt )
epoch.sd(obj, epoch.size=10, incl.date = FALSE, sqrt )
epoch.median(obj, epoch.size=10, incl.date = FALSE, sqrt )
epoch.mad(obj, epoch.size=10, incl.date = FALSE, sqrt )
epoch.autocor(obj, epoch.size=10, lag = 1, type =
    c("correlation", "covariance", "partial"), incl.date = FALSE, sqrt)
epoch.quantile(obj, epoch.size = 10,
    quantiles= c(0.1, 0.25, 0.5, 0.75, 0.9), incl.date = FALSE, sqrt )

svm(obj, sqrt )
```

Arguments

`obj`	The object to compute statistics for. Can be an "AccData" object, a matrix, or a vector.
`epoch.size`	Numeric giving intervals to consider and aggregate. For "AccData" `obj` taken as seconds. Otherwise, considered as rows, or as individual readings.
`incl.date`	logical. If TRUE, include a column of times or original indices with the results.
`FUN`	A function to be applied to each epoch.
`sqrt`	logical. If TRUE, the square rooted svm will be used in computations instead.
`lag`	Autocorrelation lag to compute.
`type`	Type of autocorrelation, as used in `acf`.
`quantiles`	Sample quantiles of SVM to compute.

Details

These functions compute epochal summary statistics for "AccData" objects, matrices and vectors.

`epoch.apply` is the general function - according to the size of `epoch.size`, it splits up the obj into collections of consecutive rows, each with the same size. These are then successively supplied to `FUN` as its first argument. If the result of FUN is a single value, then the results are concatenated into a vector output. Otherwise, an array is formed with each row corresponding to a single epochal group. For AccData, the sampling frequency of the dataset is used to interpret the epoch size in seconds. Otherwise, the raw record indices are used. If incl.date is set, the original timestamp vector of the data, or the original indices, are downsampled and included as the first column of the output.

The remaining functions are wrappers that compute various commonly useful statistics – in particular, applied to "AccData" objects and arrays, they by default compute the epochal SVM mean, standard deviation, median, median absolute deviation, and autocorrelation, and sample quantiles respectively. (Arrays are treated as each column representing the x, y, and z components respectively.) Applied to vector input, processing will occur without the SVM calculation. This behaviour may be overridden by the sqrt setting, which will force the function to use the squared (default for arrays and "AccData") or original unit (default for vectors) values in the statistical analysis.

`svm` acts identically to 'epoch.mean', with the epoch set to the sampling period. In other words, it computes the instantaneous sum of vector magnitudes of the acceleration at each record point. The function takes "AccData", array and vector input. Note that if provided with an array with 4 or more columns, columns 2 to 4 are used – the first column is regard as a timestamp and hence ignored.

Value

A vector or array giving the computed epochal summaries. With `incl.date = TRUE`, the result is given as a data.frame suitable for plotting.

See Also

`plot.AccData`, `summary.AccData`, `aggregate`, `acf`

Examples

```
dat <- read.bin(system.file("binfile/TESTfile.bin", package = "GENEAread")[1]
    , calibrate = TRUE)
```

```
#look for the epochs that exceed a certain threshold 50% of the time
plot(epoch.apply( dat, epoch.size = 3 ,
    FUN = function(t) mean(abs(svm(t) -1)>0.2)> 0.5 ), type = "l")

plot(dat[,1], svm(dat), log = "y", pch = ".")
lines(epoch.mean(dat, incl.date = TRUE), lwd = 2)
lines(epoch.mean(dat, epoch.size = 30, incl.date = TRUE), col = 2, lwd = 2)
#this should give all the same results, but by a different way
lines(epoch.apply(dat, epoch.size = 30,
    FUN = function(A) mean(svm(A, FALSE)), incl.date = TRUE), col = 3)
epsize = 30; lines(epoch.apply(dat, epoch.size = epsize,
    FUN = function(t) median(t[,1])), epoch.apply(dat, epoch.size = epsize,
    FUN = function(A) mean(svm(A, FALSE))), col = 4)
#note this is different
lines(epoch.apply(dat, epoch.size = epsize,
    FUN = function(t) median(t[,1])),epoch.apply(dat, epoch.size = epsize,
    FUN = function(A) mean(svm(A, sqrt = TRUE)))^2, col = 5)

#plot some statistics
par(mfrow = c(5,1), mar = c(1,4.5,1,1))
plot(epoch.sd(dat), type="l")
plot(epoch.median(dat), type= "l")
plot(epoch.mad(dat), type= "l")
plot(epoch.autocor(dat), type= "l")
tmp = epoch.quantile(dat, quantiles= c(0.1, 0.25, 0.5, 0.75, 0.9)); matplot(tmp, type = "l")
```

---

get.intervals *Extract an interval of data.*

Description

Function for extracting sub intervals of data, and implementation of just-in-time loading.

Usage

```
get.intervals(x, start=0, end = 1, length = NULL, time.format = c("auto", "seconds", "days", "pr
```

Arguments

`x`	Object to process. Can be array,
`start`	Start of interval.
`end`	End of interval.
`length`	Length of interval.
`time.format`	Method with which `start` and end should be understood.
`incl.date`	logical. Include a column denoting time?
`simplify`	logical. If TRUE, output an array. Otherwise output a AccData object.
`read.from.file`	logical. If TRUE, re-read the relevant time interval from the original bin file.
`size`	Desired number of samples in output.
`...`	Additional arguments to be passed to `read.bin`, if `read.from.file` is TRUE.

Details

The function extracts the desired analysis time window specified by `start` and `end`. If length is specified, then the end is set to a point `length` units after start. The times are interpreted in terms of `time.format`. For convenience, a variety of time window formats are accepted:

"seconds": Seconds since start of dataset.

"days": Days since start of dataset.

"proportion": Proportional point within dataset, given as a numeric between 0 and 1.

"measurements": Raw number of samples since start of dataset.

"time": Time string, as understood via `parse.time`.

"auto": Default - attempt to determine time format from size and type of `start`.

Some capacity for using mixed types of inputs for `start` and `length` in particular is present.

The input object `x` is typically an "AccData" object, though arrays are also accepted. "VirtAccData" are dealt with by using the timestamp and call information recorded within them to do a new read of the original bin file, assuming this is still available. This is useful for 'just in time' reads of data. "AccData" can be dealt with in this way by setting `read.from.file`.

Note that for `read.from.file`, only "time" and "proportion" `time.format` are presently supported.

Value

With `simplify = FALSE`, an "AccData" S3 object with the desired records.

Otherwise, an array containing either 3 or 4 columns, containing the x, y, z acceleration vectors and optionally a time vector.

See Also

`read.bin`, `[.AccData`, `get.intervals`

Examples

```
binfile  = system.file("binfile/TESTfile.bin", package = "GENEAread")[1]

#Read in a highly downsampled version of the file
procfile<-read.bin(binfile, downsample = 100)
print(procfile)
#Plot the x component
plot(procfile[,1:2], type = "l")

#Overlay some segments in different colour
lines(get.intervals(procfile, start = 0.4, end = 0.5, time.format = "prop", incl.date = TRUE)[,1:2], col=2
lines(get.intervals(procfile, start = 0.4, end = 5, time.format = "sec", incl.date = TRUE)[,1:2], col=3)
lines(get.intervals(procfile, start = "16:51", end = "16:52", time.format = "time", incl.date = TRUE)[,1:2
#Note that measurements will depend on the downsampling rate, not the original sampling rate of the data
lines(get.intervals(procfile, start = 100, length = 10, time.format = "measurement", incl.date = TRUE)[,1:
#This is also understood
lines(get.intervals(procfile, start = "16:52:10", 30,  incl.date = TRUE)[,1:2], col=6)

#Now load in virtually
virtfile<-read.bin(binfile, virtual = TRUE)
#Notice that get.intervals with simplify = FALSE gives a genuine AccData object
realfile = get.intervals(virtfile, start = 0.5, end = 1, simplify = FALSE)
```

```
virtfile
realfile
#get.intervals calls read.bin automatically
points(get.intervals(virtfile, start = "16:52:10", "16:52:40",  incl.date = TRUE)[,1:2], col=4, pch = ".")

#Alternatively, re-read procfile at a different resampling rate.
lines(get.intervals(procfile, start = "16:49:00", "16:49:30",  incl.date = TRUE, read.from.file = TRUE, do
```

GRtime *Date time handling for the GENEAread package.*

Description

Stores date time data as a numeric, with facility for pretty printing and axis commands.

Usage

```
convert.time(x, format = NULL)
as.GRtime(x, format = NULL, ...)
## S3 method for class 'GRtime'
format(x, format = NULL, ...)
## S3 method for class 'GRtime'
axis(side, x=NULL, at=NULL, format = NULL,labels  = TRUE, add = TRUE,  ...)
## S3 method for class 'GRtime'
pretty(x, n = 5, ...)
```

Arguments

x	Object to process. For `convert.time`, must be numeric. For `as.GRtime` may be numeric or character. For `format.GRtime`, a GRtime object, or a numeric.
format	A character string indicating the form of output. See `strptime` for details. If NULL, will be automatically chosen.
add	logical. If TRUE, actually plot the axis.
at, side, labels	Additional arguments as in `axis`.
n	Approximate number of breakpoints.
...	Additional arguments to be passed to `parse.time`, `as.numeric`, `format.POSIXct`, `axis`, `pretty.POSIXt`.

Details

The GRtime class handles dates and times for the GENEAread class. The class treats dates as numerics denoting seconds since the UNIX epoch, with potentially a string attached specifying the format to print in. Unlike `POSIXct`, we avoid some of the processing, especially with respect to time zones, and allow some more flexibility in time computation and display. A range of operators are defined.

`convert.time` converts numerics to GRtime objects. The `format` argument allows a format string to be attached specifying the default format to display in. `as.GRtime` is a wrapper to `convert.time`, that when supplied with character input, coerces the value first to numeric using `parse.time`.

`format.GRtime` formats GRtime objects for pretty printing. If `format` is provided as argument, that is used. Else, if the `format` attribute is set on `x`, that is used. Finally, if formats are not provided, and `x` is of length greater than one, the range of values of `x` is used to decide the units displayed. Numerics are also accepted - they are coerced to GRtime.

`axis.GRtime` is used to plot GRtime axis, choosing, by default, breakpoints that give 'pretty' sub intervals. Note that `plot.default` uses `axis.GRtime` by default if supplied with a GRtime object in one of the directions. However, `image.default` based functions do not use the class axis functions, so axes must be plotted manually.

`pretty.GRtime` computes 'pretty' breakpoints, using the algorithm of `pretty.POSIXt`. Attributes are preserved.

Value

For `convert.time`, `as.GRtime` and `pretty.GRtime`, a GRtime object.

For `format.GRtime` a character string representation.

For `axis.GRtime` a list containing positions and labels for axis markers.

See Also

`parse.time`, `get.intervals`, `print.AccData`

Examples

```
as.GRtime("00:01")
#format is automatically set
convert.time(1:10)
convert.time(1:10*1000)
#we add a different default format
convert.time(1:10*1000, "%H:%M:%OS3") -> t
t
str(t)
#we override format with our own
format(t, format = "%a %d/%m/%y %H:%M:%OS3")

#plot calls axis.GRtime automatically. Notice
#that the format attribute is used.
plot(t, 1:10)
#strip out the default format
t2 = convert.time(t, format = NULL)
plot(t2, 1:10)

#image plots are a bit more complex

Z = matrix(rnorm(100), 10)
image(x = t, y = t2, z = Z, axes = FALSE)
axis.GRtime(x = t2, side = 2)
Axis(x = t, side = 1) #Axis also works
box() #complete the bounding box

#custom axes
plot(t2, 1:10, xaxt = "n")
axis.GRtime(at = pretty(t2, 20) , side = 1)
```

hanning.window *Computes the Coefficients of a Hanning or Uniform Window.*

Description

For `hanning.window`, the filter coefficients w_i of a Hanning window of length `n` are computed according to the formula

$$w_i = 0.5 - 0.5 \cos \frac{2\pi i}{n-1}$$

For `uniform.window`, a constant value 1 is repeated for the length(n).

Usage

```
hanning.window(n)
uniform.window(n)
```

Arguments

`n`	The length of the window.

Value

A vector containing the filter coefficients.

Author(s)

Andreas Weingessel

References

For a definition of the Hanning window, see for example
Alan V. Oppenheim and Roland W. Schafer: "Discrete-Time Signal Processing", Prentice-Hall, 1989.

See Also

stft

Examples

```
hanning.window(10)

x<-rnorm(500)
y<-stft(x, wtype="hanning.window")
plot(y)
```

`header.info` *Get header info from GENEA output (.bin) file*

Description

Function to extract relevant header fields and values from a file.

Usage

```
header.info(binfile, more=TRUE)
```

Arguments

`binfile`	The file from which to extract the header
`more`	logical. If TRUE, extract additional data from file useful for calibration and data reading.

Details

The function extracts useful information from a .bin file, such as information about the Genea device used to produce the output, and characteristics of the subject who wore the device. The function also accepts data that has been compressed in ‘gzip’, ‘bzip2’ or ‘xz’ formats. See `file`. With `more` set to TRUE, additional data is extracted, mainly for internal use in `read.bin`.

Value

A `data.frame` with extracted header information, each row a particular header field with its value. If `more` is TRUE, an attribute "calibration" is attached to the object, consisting of a list with measurement offsets, sampling frequency estimates, start times and time zones, data position offsets, and if mmap is detected, byte locations and increments for mmap reading.

Warning

This function is specific to header structure in Geneactiv output files. By design, it should be compatible with all firmware and software versions to date (as of version of current release). If order or field names are changed in future .bin files, this function may have to be updated appropriately. The function works by looking for appropriate section headings in the .bin files.

See Also

`read.bin`

Examples

```
fileheader <- header.info(system.file("binfile/TESTfile.bin", package = "GENEAread")[1], more = TRUE)
print(fileheader)
attr(fileheader, "calibration")
```

`parse.time` *Parses a character time representation to another format.*

Description

Converts a character vector in a variety of forms into either the raw second, second classed as POSIXct, or days since Unix epoch.

Usage

```
parse.time(t="",format=c("seconds", "days", "POSIX"), tzone = 0,
start = NULL, startmidnight = NULL)
```

Arguments

`t`	A character string representation of a date-time expression.
`format`	A character string indicating which representation to output. Can be either `seconds`, `days` or `POSIX`.
`tzone`	The time zone the time is given in, expressed as an offset from UTC in hours.
`start`	Earliest allowable time stamp in the data, as seconds since Unix epoch.
`startmidnight`	Midnight of day '0' in the data, as seconds since Unix epoch.

Details

The function processes character vectors of the form "DATE TIME" – that is to say, a maximum of two terms separated by a space per value.

"TIME" is given in 24 hour format, seperated by colons, in "hh:mm", "hh:mm:ss", "hh:mm:ss:ms" or "hh:mm:ss.ms" format. If ommitted, the time is taken to be 00:00:00.000.

"DATE" can be a date representation as "YYYY-MM-DD", "DD/MM/YY" or "DD/MM/YYYY" (noting the use of a colon or backslash seperator to distinguish between the two). Alternatively, with `start` and/or `startmidnight` supplied, an integer "NN" or string "DOW" corresponding to a day of the week can be used instead. Then, the function will find the first timestamp matching the correct "TIME", that falls NN midnights after `startmidnight` and is after `start`, or, in the latter case, the first timestamp after the day of `start` that matches the appropriate day of the week. If a blank "DATE" is supplied, the function will either use the UNIX epoch, or find the first match, corresponding to the case NN = 0.

Once this is done the time is converted to the required format: `POSIX` is the usual R POSIXct format; `days` is the julian days since UNIX epoch 1970-1-1; `seconds` is the number of seconds (including subseconds) since 1970-1-1. Note that for formats other than POSIX, the output is in the same timezone as `tzone`. POSIX stores the time internally as the time in UTC, and applies a format that gives this time local to the user.

Value

A converted date-time string in the specified format. In the case of "seconds", or "days", a numeric. For POSIX, a `POSIXct` object.

See Also

`convert.time`, `get.intervals`

Examples

```
t1 = parse.time("2012-06-21 13:04:01"); print(t1)
parse.time("21/06/12 13:04:01") #gives the same result

parse.time(c("19/07/70", "20/07/70"), format = "days")
#results here will depend on your locale
parse.time(c("19/07/70", "20/07/70"), format = "POSIX", tzone = -4)

#one is the same day, one can only find a match the next day
parse.time("13:05", start = t1) - t1
parse.time("13:00", start = t1) - t1
#asking to wait 1 midnight means both times are considered as
#times on the same, full day of data
parse.time(c("1 13:05", "1 13:00"), start = t1) - t1
#2012-06-21 is a Thursday, so this is equivalent
parse.time(c("Fri 13:05", "Fri 13:00"), start = t1) - t1
#Longer form days of the week are also understood. Note that
#the first day does not get matched.
parse.time(c("Thursday 13:05", "Thursday 13:00"), start = t1) - t1
```

`plot.stft` *Plots and prints Short Time Fourier Transforms*

Description

Processes a dataset, creating an object contained processed time-frequency analyses. These can then be plotted.

Usage

```
## S3 method for class 'stft'
plot(x, mode = c("decibels", "modulus", "pval"), log = "", showmax = TRUE, median = FALSE, xaxis
## S3 method for class 'stft'
print(x, ...)
```

Arguments

`x`	"stft" class object to be processed.
`mode`	What should be plotted? "decibels": log10 of FFT modulus "modulus": Raw FFT modulus "pvalue": P-value of each frequence's modulus assuming that window was in fact white noise of equal equal standard deviation
`log`	For `log = "y"`, use a log scale on the y axis.
`showmax`	Vector or logical. Compute and plot the principle frequency components?
`median`	logical. If TRUE, smooth the STFT plot in the time direction with a running median.
`xaxis`	logical. If TRUE, plot pretty time axes.

`topthresh`	For finite values, crop plot for frequencies higher than this value, and show a summary plot up top.
`reassign`	logical. Plot reassigned stft, if available?
`xlim, ylim`	Parameters controlling axes limits of plot.
`new`	logical. If TRUE, make a new plot. Otherwise overlay on to existing plot.
`zlim.raw`	Raw values at which to threshold values for computation of heatmap colours.
`zlim.quantile`	Quantile values at which to threshold values for computation of heatmap colours.
`cex`	Size of points for reassigned STFT plotting.
`col`	Vector of colours to be used for plotting.
`...`	Additional arguments to be passed to methods.

Details

STFT objects are created by the `stft` function. These methods print some useful summary statistics about them, and produce plots.

`mode` determines the type of plot. "decibel" and "modulus" work with the raw values, while "pvalue" conducts some degree of normalisation in each time window and so is perhaps more useful for data showing a large variation in sd across different points in time. If the `null.calc` was set in the original stft argument, that is used - otherwise, an Exponential distribution is fit to each window, and the pvalues computed from that.

By default, the function uses some empirical quantile based colour thresholds designed to give somewhat reasonable and informative plots. This can be overridden, however, by setting different `zlim.raw` or `zlim.quantile` results. This can be useful for comparing two different datasets.

Reassigned stft plots are constructed, by default, when they are available, and when the original was not a "mv" stft. Unlike the heatmap used in the usual stft plot, a 2d scatterplot is used instead. This means that if there are few data points, it can be advantageous to set a higher `cex` value for larger points and better display.

With Accelerometer data, often the frequencies of interest are concentrated at the lower frequencies. Topthresh crops the frequency display to show only those frequencies. A summary plot is show on the top, to compensate. Choosing a grid of frequencies, this plot draws one line to represent the energies present in the signal at that particular frequency, and higher. Black lines are drawn for frequencies less than 2/3 the `topthresh`, red lines for 2/3 - 1 times `topthresh`, and blue lines for frequencies higher than `topthresh`. Alternative, set `log = "y"` to put frequencies on a log scale.

Value

These functions are run for their side effects.

See Also

`stft`, `image.default`

Examples

```
#Real data
binfile  = system.file("binfile/TESTfile.bin", package = "GENEAread")[1]

#Read in the entire file, calibrated
procfile<-read.bin(binfile)
```

```
#Create stft object
obj = stft(procfile, type = "svm", quiet = TRUE)
#Look at it
print(obj)

plot(obj, cex = 5)
plot(obj, showmax = FALSE, cex = 5) #suppress principals

#pval plot
plot(obj, mode = "pval", cex = 5)
#disable reassigned stft
plot(obj, mode = "pval", reassign = FALSE)
#median smoothing
plot(obj, mode = "pval", reassign = FALSE, median = TRUE)
#log scale frequency, no top bar
dev.new(); plot(obj, mode = "pval", reassign = FALSE, topthresh = Inf, log = "y")
```

read.bin *File processing function for binary files.*

Description

A function to process binary accelerometer files and convert the information into R objects.

Usage

```
read.bin(binfile, outfile = NULL, start = NULL, end = NULL,
    verbose = TRUE, do.temp = TRUE,do.volt = TRUE, calibrate = TRUE, downsample = NULL, blocksiz
```

Arguments

binfile	A filename of a file to process.
outfile	An optional filename specifying where to save the processed data object.
start	Either: A representation of when in the file to begin processing, see Details.
end	Either: A representation of when in the file to end processing, see Details.
verbose	A boolean variable indicating whether some information should be printed during processing should be printed.
do.temp	A boolean variable indicating whether the temperature signal should be extracted.
do.volt	A boolean variable indicating whether the voltage signal should be extracted.
calibrate	A boolean variable indicating whether the raw accelerometer values and the light variable should be calibrated according to the calibration data in the headers.
downsample	A variable indicating the type of downsampling to apply to the data as it is loaded. Can take values: `NULL`: (Default) No downsampling Single numeric: Reads every `downsample`-th value, starting from the first. Length two numeric vector: Reads every `downsample[1]`-th value, starting from the `downsample[2]`-th.

Non-integer, or non-divisor of 300 downsampling factors are allowed, but will lead to imprecise frequency calculations, leap seconds being introduced, and generally potential problems with other methods. Use with care.

`blocksize`	Integer value giving maximum number of data pages to read in each pass. Defaults to 10000 for larger data files. Sufficiently small sizes will split very large data files to read chunk by chunk, reducing memory requirements for the read.bin function (without affecting the final object), but conversely possibly increasing processing time. Can be set to Inf for no splitting.
`virtual`	logical. If set TRUE, do not do any actual data reading. Instead construct a VirtualAccData object containing header information to allow use with `get.intervals`.
`mmap.load`	logical. If TRUE (Default on 64bit R), use the `mmap` package to process the binfile.
`pagerefs`	A variable that can take two forms, and is considered only for `mmap.load = TRUE`

NULL or FALSE, in which case pagerefs are dynamically calculated for each record. (Default)
A vector giving sorted byte offsets for each record for mmap reading of data files.
TRUE, in which case a full page reference table is computed before any processing occurs.

Computing pagerefs takes a little time and so is a little slower. However, it is safer than dynamic computations in the case of missing pages and high temperature variations. Further, once page references are calculated, future reads are much faster, so long as the previously computed references are supplied.

`...`	Any other optional arguments can be supplied that affect manual calibration and data processing. These are:

`gain`: a vector of 3 values for manual gain calibration of the raw (x,y,z) axes. If `gain=NULL`, the gain calibration values are taken from within the output file itself.

`offset`: a vector of 3 value for manual offset calibration of the raw (x,y,z) axes. If `offset=NULL`, the offset calibration values are taken from within the output file itself.

`luxv`: a value for manual lux calibration of the light meter. If `luxv=NULL`, the lux calibration value is taken from within the output file itself.

`voltv`: a value for manual volts calibration of the light meter. If `voltv=NULL`, the volts calibration value is taken from within the output file itself.

`warn`: if set to true, give a warning if input file is large, and require user confirmation.

Details

The read.bin package reads in binary files compatible with the GeneActiv line of Accelerometers, for further processing by the other functions in this package. Most of the default options are those required in the most common cases, though users are advised to consider setting start and end to smaller intervals and/or choosing some level of downsampling when working with data files of longer than 24 hours in length.

The function reads in the desired analysis time window specified by `start` and `end`. For convenience, a variety of time window formats are accepted:

Large integers are read as page numbers in the dataset. Page numbers larger than that which is available in the file itself are constrained to what is available. Note that the first page is page 1.

Small values (between 0 and 1) are taken as proportions of the data. For example, ‘start = 0.5‘ would specify that reading should begin at the midpoint of the data.

Strings are interpreted as dates and times using `parse.time`. In particular, times specified as "HH:MM" or "HH:MM:SS" are taken as the earliest time interval containing these times in the file. Strings with an integer prepended, using a space seperator, as interpreted as that time after the appropriate number of midnights have passed - in other words, the appropriate time of day on the Nth *full* day. Days of the week and dates in "day/month", "day/month/year", "month-day", "year-month-day" are also handled. Note that the time is interpreted in the same time zone as the data recording itself.

Actual data reading proceeds by two methods, depending on whether `mmap` is true or false. With `mmap = FALSE`, data is read in line by line using `readLine` until blocksize is filled, and then processed. With `mmap = TRUE`, the `mmap` package is used to map the entire data file into an address file, byte locations are calculated (depending on the setting of `pagerefs`), `blocksize` chunks of data are loaded, and then processed as raw vectors.

There are advantages and disadvantages to both methods: the mmap method is usually much faster, especially when we are only loading the final parts of the data. ReadLine will have to process the entire file in such a case. On the other hand, mmap requires a large amount of memory address space, and so can fail in 32 bit systems. Finally, reading of compressed bin files can only be done with the readLine method. Generally, if mmap reading fails, the function will attempt to catch the failure, and reprocess the file with the readLine method, giving a warning.

Once data is loaded, calibration is then either performed using values from the binary file, or using manually inputted values (using the `gain`, `offset`,`luxv` and `voltv` arguments).

Value

With `virtual = FALSE`, an "AccData" S3 object with 9 components:

`data.out`	A 6 or 7 column matrix of the processed pages, the rows of which are the processed observations in order of processed pages. The matrix has columns (timestamp,x-axis,y-axis,z-axis,light,button) or (timestamp,x-axis,y-axis,z-axis,light,button,temperat if `do.temp=TRUE`. The timestamp is stored as seconds since 1 Jan 1970, in the timezone that the data is recorded in.
`page.timestamps`	The timestamps as POSIXct representations (as opposed to those within the `data.out` array.)
`freq`	The effective sampling frequency (in Hz).
`filename`	The file name of the bin file.
`page.numbers`	The pages that were loaded.
`call`	The function call that the object was created with.

`page.volts`	The battery voltage associated with each loaded page, if `do.volt` is TRUE.
`pagerefs`	The page byte offsets that were computed.
`header`	File header output, as given by `header.info`.

Various processing methods are implemented so that `AccData` objects can be treated as an ordinary matrix in many cases. See `print.AccData` for info.

With `virtual = TRUE`, a "VirtAccData" S3 object with page.timestamps, freq, filename, page.numbers, call, pagerefs, header as in the earlier case, but also,

`data.out`	A vector containing the timestamps of each page, using local seconds since 1970.
`nobs`	Number of observations per page, after downsampling.

Warning

Reading in an entire .bin file will take a long time if the file contains a lot of datasets. Reading in such files without downsampling can use up all available memory. See `memory.limit`.

This function is specific to header structure in Geneactiv output files. By design, it should be compatible with all firmware and software versions to date (as of version of current release). If order or field names are changed in future .bin files, this function may have to be updated appropriately.

See Also

`header.info`, `print.AccData`, `get.intervals`

Examples

```
binfile  = system.file("binfile/TESTfile.bin", package = "GENEAread")[1]

#Read in the entire file, calibrated
procfile<-read.bin(binfile)
print(procfile)
procfile$data.out[1:5,]

#Uncalibrated, mmap off
procfile2<-read.bin(binfile, calibrate = FALSE)
procfile2$data.out[1:5,]

#Read in again, reusing already computed mmap pagerefs
procfile3<-read.bin(binfile, pagerefs = procfile2$pagerefs )

#Downsample by a factor of 10
procfilelo<-read.bin(binfile, downsample = 10)
print(procfilelo)
object.size(procfilelo) / object.size(procfile)

#Read in a 1 minute interval
procfileshort <- read.bin(binfile, start = "16:50", end = "16:51")
print(procfileshort)

##NOT RUN: Read, and save as a R workspace
#read.bin(binfile, outfile="tmp.Rdata")
#print(load("tmp.Rdata"))
#print(processedfile)
```

stft — *Computes Short Time Fourier Transforms*

Description

Processes a dataset, creating an object contained processed time-frequency analyses. These can then be plotted.

Usage

```
stft(X, start=0, end=1, length=NULL,  time.format = c("auto"),
    type = c("mv", "svm", "sum"), mv.indices, date.col,
    reassign = TRUE,plot.it = FALSE,...)
```

Arguments

`X`	The dataset to be processed.
`start, end, length, time.format`	A specification for the segment to process, as in `get.intervals`.
`type`	The type of STFT to compute.
`mv.indices`	For `type = "mv"` or `type = "sum"`, the indices to process and the order to process them in.
`date.col`	logical. Whether the first column should be ignored and treated as a timestamp. If unset, is automatically chosen.
`reassign`	logical. If TRUE, compute the time-reassigned STFT. For `type %in% c("mv", "sum")`, this is done with the first coordinate in `mv.indices`.
`plot.it`	logical. Whether to plot the STFT immediately when processing is complete, using the default `plot.stft` options.
`...`	Additional optional arguments to control the STFT computation. These are: `win`: Window size in seconds for STFT computation. Increased window size mean better frequency resolution, but poorer time resolution. Defaults to 10 seconds. `inc`: Increment between successive time steps for processing. Defaults to `win/2`. `coef`: Number of fourier frequencies to compute. Small values will remove the higher frequencies from the processed object. Defaults to the maximum, `win/2`. `wtype`: String giving the name of a window function, providing coefficients for filtering before processing. "hanning.window" is the default, with "uniform.window" also available.

`freq`: Sampling frequency of data set. If not given, is taken from X itself, or assumed to be 1 if unavailable.

`center`: If TRUE (Default), center the data in each window before processing is done. Useful for avoiding excessively large DC offset coefficients in results.

`calc.null`: If TRUE (Defaults to FALSE), compute a 'null' STFT by resampling the data completely, then doing a STFT.

`pvalues`: If TRUE (Defaults to FALSE) Compute bootstrapped pvalues for each position by resampling within each window and applying a wilcox test.

`time`: Allows the user to set an overriding timestamp vector to be used for processing.

`quiet`: If TRUE, suppress output.

Details

This function accepts input in a variety of forms and computes short time fourier transforms to extract frequency structure from the data.

X may be an array, a vector, or an AccData object. If date.col is TRUE, the first column of an array X would be used to determine timestamps. Otherwise indices would be used. If date.col is not set, the function will attempt to determine whether the first column is timestamp-like. The timestamp column is removed from X (and so not included in consideration of `mv.indices`, for instance).

With vectors, the basic method is to compute the STFT by creating windows of size `win` seconds every `inc` seconds, and computing the fourier transform. With multi-dimensional data and AccData, processing is done on the dimensions that are in `mv.indices`, or the first three non-date columns if that is unavailable. Three methods are possible:

1. `type = "mv"`: The one dimensional method is first applied to each of the chosen column indices. These are then collated by choosing, for each time-frequency combination, the maximum such value across each of the indices.

2. `type = "svm"`: The SVM is computed first for each time step by computing the square rooted sum of squares. This is then dealt with using the one dimensional method.

3. `type = "sum"`: As in "mv", the 1d method is applied. The square of the modulus of the result is then summed and square rooted.

If `reassign` is set, the time reassigned stft is also computed for the first element of `mv.indices` or the svm as appropriate, by using finite differencing. This gives potentially better resolution results for data with a clear signal component.

Value

A "stft" class object - a list with the following components:

`call`	The function call.
`type`	Type of STFT computed.
`values`	Mod of FFT computed, with each row corresponding to a specific time increment.
`increment, windowsize, center, sampling.frequency`	Various control parameters used in the computation.

`null.logmean, null.logsd`	Log of the square rooted mean and standard deviation of the Mod FFT squared for the randomised data, if `calc.null = TRUE`.
`p.values`	Wilcoxian pvalues, if `pvalues = TRUE`.
`principals`	Principal frequencies.
`frequency`	Frequencies at which FFT is computed.
`time`	Timestamps for FFT windows.
`LGD`	Local group delay matrix for reassigned STFT.
`CIF`	Channelized instantaneous frequency matrix for reassigned STFT.

Acknowledgements

The initial implementation of this function is based on that in package `e1071`. The reassigned STFT implementation is due to Nelson (2001), via Fulop (2006).

References

Fulop, S.A. & Fitz, K. (2006). Algorithms for computing the time-corrected instantaneous frequency (reassigned) spectrogram, with applications *J Acoustical Society of America* **119(1)**, 360–371.

Nelson. D.J. (2001). Cross-spectral methods for processing speech *J Acoustical Society of America* **110(1)**, 2575-2592.

See Also

`plot.stft`, `get.intervals`

Examples

```
#Some artificial data
time = 1:5000
#sum of two sine curves at 0.3 Hz and 0.05 Hz
f1 = 0.3; f2 = 0.05
sin1 = sin(time * f1 * 2*pi)
sin2 = sin(time * f2 * 2*pi)
#add a bit of noise
signal = sin1 + sin2 + 1*rnorm(5000)
#non-reassigned
stft(signal, plot = TRUE, reassign = FALSE, win = 100)
#reassigned
stft(signal, plot = TRUE, reassign = TRUE, win = 100)

#add a third component: varying frequency.
stft(signal + sin( cumsum(seq(f2, f1, length = 5000))*2*pi), plot = TRUE, reassign = TRUE, win = 100)

#Real data
binfile  = system.file("binfile/TESTfile.bin", package = "GENEAread")[1]

#Read in the entire file, calibrated
procfile<-read.bin(binfile)
#Default is mv
stft(procfile, plot.it = TRUE)
#Try sum?
```

```
stft(procfile, plot.it = TRUE, type = "sum", reassign = FALSE)

#Just look at the last 50% of the data
stft(procfile, start = 0.5, plot.it = TRUE)

#not reassigned, svm
stft(procfile, type = "svm", reassign = FALSE, plot.it = TRUE)
#a narrower 5 second window means better time resolution
stft(procfile, type = "svm", reassign = FALSE, plot.it = TRUE, win = 5)
#choose increments so as not to overlap
stft(procfile, type = "svm", reassign = FALSE, plot.it = TRUE, win = 5, inc = 5)
#uniform windows
stft(procfile, type = "svm", reassign = FALSE, plot.it = TRUE, wtype = "uniform.window")

#Svm, reassigned, quietly
obj = stft(procfile, type = "svm", quiet = TRUE)
plot(obj, cex = 3, showmax = FALSE, mode = "pval")
```

Index