Package 'particleShearEvaluation'

March 10, 2021

Type Package

Title Import and analysis of output files from the python simulation particleShear

Version 1.0

Date 2018-10-17

Author Thomas Braschler

Maintainer Thomas Braschler <thomas.braschler@gmail.com>

Description This R package allows reading and analyzing data files produced by the particleShearSimulation Python library.

License GPL-3 + file LICENSE

Depends rsq, R (¿= 3.5.0), methods, MASS

LazyLoad yes

R topics documented:

particleShearEvaluation-package . . . 2
evaluate_stress_curve . . . 2
file_information_table_from_folder . . . 3
file_information_table_from_folders . . . 3
find_infos_from_file . . . 4
find_info_from_file . . . 5
find_info_from_text . . . 5
general_linear_regression_p_bootstrap . . . 6
get_yield_point . . . 13
G_from_demodulation . . . 14
headerSearchPatterns . . . 15
linear_regression_p_bootstrap . . . 16
measurement_periods . . . 23
read_first_lines . . . 24
read_general_data_from_file . . . 24
read_stress_curve_from_file . . . 25
read_stress_curve_from_rda_file . . . 26
read_stress_tensor_data_from_file . . . 27
read_stress_tensor_data_from_rda_file . . . 29
split_tensor_and_overview_data . . . 30
stress_data_start_line . . . 31

Index **32**

`particleShearEvaluation-package` *plot.counts*

Description

Utility functions to read text output files from the simulation library particleShearEvaluation (Python)

Details

Package:	particleShearEvaluation
Type:	Package
Version:	1.0
Date:	2018-10-17
License:	What license is it under?
LazyLoad:	yes

Author(s)

Thomas Braschler

Maintainer: thomas.braschler@gmail.com

`evaluate_stress_curve` *evaluate_stress_curve*

Description

Evaluate oscillatory sweep curve (G' and G" as a function of amplitude), looking for a low strain plateau, a first softening transition, a soft plateau, and a yield transition

Usage

```
evaluate_stress_curve(strain,Gprime,Gprimeprime,guess_softening_strain=0.01,guess_soft_plateau_s
```

Arguments

`strain`	Strain, should be vector in ascending strain order
`Gprime`	Gprime values, same length as strain
`Gprimeprime`	G" values, should be the same length as previous arguments
`guess_softening_strain`	Initial guess for the softening strain, typically the same as the friction coefficient
`guess_soft_plateau_strain`	Guess of the strain on the soft plateau, typically 0.1

Value

A matrix of 3 rows and 4 columns. First row is strain, second row is G', third row is G". The columns are the low strain plateau, the softening transition, the soft plateau, and the yield transition

Author(s)

Thomas Braschler

`file_information_table_from_folder`

Search a folder for text files to retrieve the standard header information from the particleShearSimulation output text files

---

Description

Finds text files of at least `min_file_size` bytes in size, reads the first 100 lines (or all available lines if the file is shortr) for the header information, and looks for the standard information by means of regular expression patterns given by headerSearchPatterns

Usage

```
file_information_table_from_folder(path,min_file_size=1e5)
```

Arguments

`path`	Folder path
`min_file_size`	Minimal size for a file to be taken into account in the listing

Value

A data frame, one line per conforming file, columns for each information element

Author(s)

Thomas Braschler

`file_information_table_from_folders`

Search several folders for text files to retrieve the standard header information from the particleShearSimulation output text files

---

Description

Finds text files of size of at least `min_file_size` bytes, reads the first 100 lines for the header information (or the entire file if less than 100 lines are found), and looks for the standard information by means of regular expression patterns given by headerSearchPatterns

Usage

```
file_information_table_from_folders(paths,min_file_size=1e5)
```

Arguments

paths	Vector of folder paths
min_file_size	Minimal size for a file to be taken into account in the listing

Details

For each element in paths, this function invokes file_information_table_from_folders, and then concatenates the results

Value

A data frame, one line per conforming file, columns for each information element

Author(s)

Thomas Braschler

---

find_infos_from_file *find_infos_from_file*

Description

Searches a text file for the information elements specified by a vector of specific regular expression patterns

Usage

```
find_infos_from_file(path,patterns)
```

Arguments

path	Path to the text file
patterns	Patterns allowing to retrieve the desired bits of information in the text file. The patterns needs are a vector, each element must have a named capturing parenthesis. An example would be c("Damping factor = (?<capture>[0-9]*\.?[0-9]*)" where the capturing name <capture> is mandatory.

Author(s)

Thomas Braschler

find_info_from_file *find_info_from_file*

Description

Find a particular information from a text file

Usage

```
find_info_from_file(path,pattern,n=100)
```

Arguments

path	Text in which to search for the regular expression pattern
pattern	Pattern allowing to retrieve a particular information in the text. The pattern needs to have a named capturing parenthesis. An example would be "Damping factor = (?<capture>[0-9]*\.?[0-9]*)", where the capturing name <capture> is mandatory.
n	Number of lines to read while searching for the information. Provide n=-1 for reading the entire file

Author(s)

Thomas Braschler

find_info_from_text *find_info_from_text*

Description

Find particular information from the file header text by using a supplied regular expression pattern

Usage

```
find_info_from_text(txt,pattern)
```

Arguments

txt	Text in which to search for the regular expression pattern
pattern	Pattern allowing to retrieve a particular information in the text. The pattern needs to have a named capturing parenthesis. An example would be "Damping factor = (?<capture>[0-9]*\.?[0-9]*)", where the capturing name <capture> is mandatory.

Author(s)

Thomas Braschler

general_linear_regression_p_bootstrap
general_linear_regression_p_bootstrap

Description

Evaluation of a bootstrapped (resampled) dataset with a glm model to obtain a p-value.

Usage

```
general_linear_regression_p_bootstrap(x,y,n_agg=5,family=gaussian(link="identity"),na.rm=FALSE,.
```

Arguments

x	Vector with numerical values associated with experimental conditions (regressor), these are the x-values for the glm model
y	Matrix obtained by the resampling procedure. y must have as many rows as there are values in x; each column corresponds to a redrawn sample. Sequential blocks of n_agg columns will be evaluated together in a glm, so ideally, the number of columns in y is a multiple of n_agg. Otherwise only the complete blocks in y are taken into account.
n_agg	Length of the resampling blocks for accross the columns of y, the will be dim(y)[2]/n_agg evaluations of the glm.
family	Family argument for the glm model, see also family
na.rm	In random resampling, it may happen that non-evaluatable samples are generated (all values identical, for example). Although generally rare, in large bootstrapping studies, it may be advantages to pass na.rm=TRUE so that these rare samples are not taken into account.
...	Additional parameters to be passed to glm

Details

Generally, this function is useful in a limited scenario where one has a procedure that best estimates parameters from average curves, but by doing so looses information on variability. If there are many x values, then this is less of problem, the issue is more if there are only a few x-values because then despite an originally potentially very large dataet, only a few x-y pairs result. The result is then that despite a potentially very large dataset used, the standard linear regression (lm) or even general linear model statistics (glm) have almost no power to detect even dramatic effects. In that case, subsampling

Value

Numerical p-value for a significant link between y and x. In addition, the following attributes are attached to this p-values p:

attr(p,"F") Average value of the t-statistics squared for the length(y)/n_agg evaluations of glm. The t-statistics are obtained from the coefficients of the summary of the glms (i.e. from summary.glm. The F-value is the mean of the squared t-values.

`attr(p,"DF1")` Degrees of freedom of the F-statistics. This is 1, since these are obtained from t-values which only have 1 degree of freedom in the numerator

`attr(p,"DF2")` Mean degree of freedom for the residual variance for evaluation of the p-value

`attr(p,"adj.r.squared")` Adjusted R2 value.

`attr(p,"p_shapiro")` Shapiro Wilks P-value for the residuals

`attr(p,"confint")` 95% confidence interval for the regression slope coefficient. In lm, this is really a slope, in glm regression, this depends on the link functions, but still, if significant, 0 should not be in this interval

Author(s)

Thomas Braschler

Examples

```
# Example 1: usage
x=c(0,1,2,3)

# With a known effect
y=matrix(nrow=length(x),ncol=500,data=x)
y=y+matrix(nrow=dim(y)[1], ncol=dim(y)[2], data=rnorm(dim(y)[1]*dim(y)[2]))
general_linear_regression_p_bootstrap(x,y,n_agg=5)

# Random effect only
y=matrix(nrow=dim(y)[1], ncol=dim(y)[2], data=rnorm(dim(y)[1]*dim(y)[2]))
general_linear_regression_p_bootstrap(x,y,n_agg=5)


# Example 2:
# ================================================================================================================

# Use of a more reality-inspired scenario: The problem of evaluation of outcomes from an averaged curve.
# This is inspired from an Arrhenius-type equation, we use here the temperature at half-maximal reaction rate fo
# assessing catalyst activity (assuming Arrhenius-type saturation at higher temperature).

# T is the known temperature, in Kelvin; T0 the activation energy in temperature equivalents
# Let's assume T0 depends on the addition of some catalyst, this is our x. Let's have the dependency quite extrem
x=c(0,3,6) # Concentrations catalyst
T0=exp(x) # Let's say, this is an exponential dependency
T=c(1,3,10,30,100,300,1000) # Known temperatures
N_per_condition=15
# Theoretical values
theory_reaction_rate = matrix(nrow=length(x),ncol=length(T))
# In reality, we would dispose of measured values with variability rather than theoretical values. Let's say we c
measured=data.frame(T=vector(mode="numeric",length=0),x=vector(mode="numeric",length=0),measured_reaction_r
for(ind_x in 1:length(x))
{
for(ind_T in 1:length(T))
{
theory_reaction_rate[ind_x,ind_T]=50*exp(-T0[ind_x]/T[ind_T])
# Let's say we have normal distribution at log scale
# Generate a random sample with 15 values
```

```
measurements_random =exp(log(theory_reaction_rate[ind_x,ind_T])+rnorm(N_per_condition))
measured=rbind(measured,data.frame(T=T[ind_T],x=x[ind_x],measured_reaction_rate=measurements_random))

}
}
plot(measured_reaction_rate ~ T, measured[measured$x==x[1],],pch=21,col="black",bg="black",ylim=c(-10,150))
for(ind_x in 2:length(x))
{
lines(measured_reaction_rate ~ T, measured[measured$x==x[ind_x],],pch=21,col=palette()[ind_x],bg=palette()[

}

# For the evaluation, let's first evaluate things globally. Let's say we are interested in knowing for each catal

v_max = aggregate ( measured_reaction_rate ~ x, measured[measured$T==max(measured$T),],FUN=median,na.rm=TRUE)
# General aggregation per x and T
v = aggregate(measured_reaction_rate ~ x+T,measured,FUN=median,na.rm=TRUE)
for(ind_x in 1:length(x))
{
lines(measured_reaction_rate ~ T, v[v$x==x[ind_x],],col=palette()[ind_x],type="l")
}


# For convenience, define here a function that estimates the temperatures for half-maximal reaction rate

estimate_half_temperature_at_half_maximal_rate<-function(v)
{
x=sort(unique(v$x))
v_max=v[v$T==max(v$T),]

# For each concentration, to find the temperature where the rate is half-maximal, we go through the temperatures
half_temperature = vector(mode="numeric",length=length(x))
names(half_temperature)=x

for(ind_x in 1:length(x))
{
vx = v[v$x==x[ind_x],]
first_point_above = min(which(vx$measured_reaction_rate>v_max$measured_reaction_rate[v_max$x==x[ind_x]]/2))
if(is.infinite(first_point_above))
{
half_temperature[as.character(x[ind_x])]=max(vx$T)
} else {
if(first_point_above==1)
{
half_temperature[as.character(x[ind_x])]=min(vx$T)
} else {
half_temperature[as.character(x[ind_x])]=approx(vx$measured_reaction_rate[c(first_point_above-1,first_point
vx$T[c(first_point_above-1,first_point_above)],xout=v_max$measured_reaction_rate[v_max$x==x[ind_x]]/2)$y
}
}

}

overall_summary = data.frame(x=x, half_temperature=half_temperature)

return(overall_summary)
```

```
}

overall_summary=estimate_half_temperature_at_half_maximal_rate(v)




dev.new()

# As one can see, it is quite difficult to infer much from this data. Even though the difference in half-maximal t
# three experimental points does not allow to conclude much
plot(half_temperature~x, overall_summary)
# The linear model is hardly ever significant
summary(lm(half_temperature~x, overall_summary))
# If you run this a couple of times, you will see that the glm with the anticipated variance does better, but stil
summary(glm(half_temperature~x, overall_summary,family=quasi(link = "log", variance = "mu")))


# That's where bootstrapping can offer a reasonable alternative
n_agg = 3 # Let's generate three subsampling plots
N_total = n_agg*100 # 100 blocks
n_to_sample = round(N_per_condition/n_agg) # Nominal coverage of 1

y=matrix(nrow=length(x),ncol=N_total)

for(ind_bootstrap in 1:N_total)
{

measured_subsampled=measured[FALSE,]

for(ind_x in 1:length(x))
{
for(ind_T in 1:length(T))
{
to_subsample = measured[measured$T==T[ind_T] & measured$x==x[ind_x],]
measured_subsampled=rbind(measured_subsampled,to_subsample[sample(dim(to_subsample)[1], n_to_sample),])

}
}

v_subsampled=aggregate(measured_reaction_rate ~ x+T,measured_subsampled,FUN=median)

half_temperature_estimate=estimate_half_temperature_at_half_maximal_rate(v_subsampled)

y[match(half_temperature_estimate$x,x),ind_bootstrap]=half_temperature_estimate$half_temperature

}

# x is now the regressor values (catalyst concentrations) as wanted

general_linear_regression_p_bootstrap(x,y,n_agg=n_agg,family=quasi(link = "log", variance = "mu"))
```

```
# Example 3: Power (1-beta) and false-positives (alpha) analysis in the setting of example 1
# ======================================================================
# Basic settings as in example 1
# as before, to get the temperature estimates at half-maximum rate empirically

# This example takes a very long time to execute (15' ), typical results are:
# False positive rates (alpha) under the hypothesis of random fluctations only:
#    => a few percent (generally compatible with alpha <= 0.05) for all three evaluation methods (simple linear m
# general_linear_regression_p_bootstrap
#    => Power (true positives, under the hypothetical true effect detailed in the example: Very low for simple li
#           a simple glm (on the order of 0.3-0.4); and moderately high (0.85-0.95) for  the bootstrapping appro
# For the particular case in this example, we therefore conclude that the bootstrapping approach is the most powe
# type I (false negatives) errors

## Not run:

estimate_half_temperature_at_half_maximal_rate<-function(v)
{
x=sort(unique(v$x))
v_max=v[v$T==max(v$T),]

# For each concentration, to find the temperature where the rate is half-maximal, we go through the temperatures
half_temperature = vector(mode="numeric",length=length(x))
names(half_temperature)=x

for(ind_x in 1:length(x))
{
vx = v[v$x==x[ind_x],]
first_point_above = min(which(vx$measured_reaction_rate>v_max$measured_reaction_rate[v_max$x==x[ind_x]]/2))
if(is.infinite(first_point_above))
{
half_temperature[as.character(x[ind_x])]=max(vx$T)
} else {
if(first_point_above==1)
{
half_temperature[as.character(x[ind_x])]=min(vx$T)
} else {
half_temperature[as.character(x[ind_x])]=approx(vx$measured_reaction_rate[c(first_point_above-1,first_point
vx$T[c(first_point_above-1,first_point_above)],xout=v_max$measured_reaction_rate[v_max$x==x[ind_x]]/2)$y
}
}

}

overall_summary = data.frame(x=x, half_temperature=half_temperature)

return(overall_summary)


}



get_p_value_estimate<-function(x,T0,T,N_per_condition)
{
```

```
theory_reaction_rate = matrix(nrow=length(x),ncol=length(T))
# In reality, we would dispose of measured values with variability rather than theoretical values. Let's say we c
measured=data.frame(T=vector(mode="numeric",length=0),x=vector(mode="numeric",length=0),measured_reaction_r
for(ind_x in 1:length(x))
{
for(ind_T in 1:length(T))
{
theory_reaction_rate[ind_x,ind_T]=50*exp(-T0[ind_x]/T[ind_T])
# Let's say we have normal distribution at log scale
# Generate a random sample with 15 values
measurements_random =exp(log(theory_reaction_rate[ind_x,ind_T])+rnorm(N_per_condition))
measured=rbind(measured,data.frame(T=T[ind_T],x=x[ind_x],measured_reaction_rate=measurements_random))

}
}

# For the evaluation, let's first evaluate things globally. Let's say we are interested in knowing for each catal

v_max = aggregate ( measured_reaction_rate ~ x, measured[measured$T==max(measured$T),],FUN=median,na.rm=TRUE)
# General aggregation per x and T
v = aggregate(measured_reaction_rate ~ x+T,measured,FUN=median,na.rm=TRUE)



overall_summary=estimate_half_temperature_at_half_maximal_rate(v)



pvals=vector(length=3, mode="numeric")

names(pvals)=c("lm simple","glm simple","glm bootstrapping")

pvals["lm simple"]=coefficients(summary(lm(half_temperature~x, overall_summary)))["x","Pr(>|t|)"]
pvals["glm simple"]=coefficients(summary(glm(half_temperature~x, overall_summary,family=quasi(link = "log",



n_agg = 3 # Let's generate three subsampling plots
N_total = n_agg*100 # 100 blocks
n_to_sample = round(N_per_condition/n_agg) # Nominal coverage of 1

y=matrix(nrow=length(x),ncol=N_total)

for(ind_bootstrap in 1:N_total)
{

measured_subsampled=measured[FALSE,]

for(ind_x in 1:length(x))
{
for(ind_T in 1:length(T))
{
to_subsample = measured[measured$T==T[ind_T] & measured$x==x[ind_x],]
measured_subsampled=rbind(measured_subsampled,to_subsample[sample(dim(to_subsample)[1], n_to_sample),])

}
}
```

```
v_subsampled=aggregate(measured_reaction_rate ~ x+T,measured_subsampled,FUN=median,na.rm=TRUE)

half_temperature_estimate=estimate_half_temperature_at_half_maximal_rate(v_subsampled)

y[match(half_temperature_estimate$x,x),ind_bootstrap]=half_temperature_estimate$half_temperature

}

# x is now the regressor values (catalyst concentrations) as wanted

pvals["glm bootstrapping"]=general_linear_regression_p_bootstrap(x,y,n_agg=n_agg,family=quasi(link = "log",

return(pvals)

}


# First, the case with a known effect
x=c(0,2,4) # Concentrations catalyst
T0=exp(x) # Let's say, this is an exponential dependency
T=c(1,3,10,30,100,300,1000) # Known temperatures
N_per_condition=15
N_simulation=100


pfirst=get_p_value_estimate(x,T0,T,N_per_condition)

p_matrix=matrix(ncol=length(pfirst),nrow=N_simulation)

p_matrix[1,]=pfirst

colnames(p_matrix)=names(pfirst)

for(ind in 2:(dim(p_matrix)[1]))
{
p_matrix[ind,]=get_p_value_estimate(x,T0,T,N_per_condition)

cat(paste("Run ", ind, "\n", sep=""))

}


power=pfirst
for(theApproach in names(power))
{
current_data=p_matrix[,theApproach]
current_data=current_data[!is.na(current_data)]
power[theApproach]=sum(current_data<=0.05)/(length(current_data))
}


# Second, false positives
T0=exp(0*x) # No effect of the catalyst


pfirst=get_p_value_estimate(x,T0,T,N_per_condition)
```

```
p_matrix_control=matrix(ncol=length(pfirst),nrow=N_simulation)

p_matrix_control[1,]=pfirst

colnames(p_matrix_control)=names(pfirst)

for(ind in 2:(dim(p_matrix_control)[1]))
{
p_matrix_control[ind,]=get_p_value_estimate(x,T0,T,N_per_condition)

cat(paste("Run ", ind, "\n", sep=""))

}


false_positives=pfirst
for(theApproach in names(false_positives))
{
current_data=p_matrix_control[,theApproach]
current_data=current_data[!is.na(current_data)]
false_positives[theApproach]=sum(current_data<=0.05)/(length(current_data))
}

for_barplot=matrix(nrow=2,ncol=length(false_positives),data=c(false_positives, power),byrow=TRUE)

rownames(for_barplot)=c("False positives (alpha)", "True positives (1-beta, power)")
colnames(for_barplot)=names(false_positives)

barplot(for_barplot,beside=TRUE,legend=TRUE)


## End(Not run)
```

---

get_yield_point *get_yield_point*

Description

Yield point by crossing of G' and G" curves in a strain vs. G' and G" oscillatory sweep diagram

Usage

```
get_yield_point(strain,Gprime,Gprimeprime)
```

Arguments

strain	Vector with strain values (meant to be imposed in the simulations), should be unique values
Gprime	Vector with G' values, identical in length to strain
Gprimeprime	Vector with G" values, identical in length to strain

Details

This is a relatively elementary function to find the intersection of the G' and G" curves. It proceeds as follows: If G" is higher than G' everywhere, the lowest strain value is reported; if on the contrary, G' is higher than G" everywhere, NA is reported. In the nominal case, the highest strain at which G' crosses below G" is reported.

Value

Strain at which the G' and G" curves intersect

Author(s)

Thomas Braschler

See Also

evaluateYield in package rheologyEvaluation (https://github.com/tbgitoo/rheologyEvaluation) with more flexible options for more complicated cases

Examples

```
strain=c(0.01,0.03,0.1,0.3,1,3,10)
Gprime=c(1000,950,700,300,100,20,10)
Gprimeprime=c(50,60,80,100,120,110,50)
plot(Gprime ~ strain,type="b",ylab=c("G' and G''"))
lines(Gprimeprime ~ strain)
lines(rep(get_yield_point(strain,Gprime,Gprimeprime),2),c(0,max(Gprime)),lty=2)
get_yield_point(strain,Gprime,Gprimeprime)
```

G_from_demodulation *G_from_demodulation*

Description

G' and G" by demodulation of a time - shear stress curve

Usage

```
G_from_demodulation(t,Frequency,m,correct_for_jumps=TRUE,Strain_amplitude=1,relative_threshold=0
```

Arguments

`t`	Vector with the time points
`Frequency`	Frequency of applied oscillatory shear
`m`	Vector with the shear stress measurements, should have same length as `t`
`correct_for_jumps`	Possibility to correct for sudden jumps, arising typically as individual spheres transit across the boundary under Lees-Edwards conditions
`Strain_amplitude`	Amplitude of the applied sinusoidal strain
`relative_threshold`	Threshold relative to minimum to maximum amplitude found in the measurement `m`
`doPlot`	Plot the jump-corrected curve?
`plot.new`	Start a new plot? Only relevant, if `doPlot=TRUE`
`max_plot`	Maximum number of values to plot for the stress curve. Only relevant if `doPlot=TRUE` and if `max_plot<length(t)`
`...`	Additional parameters to be passed to the graphical functions (either plot if `doPlot=TRUE` or otherwise lines

Details

The function assumes that the first time point is at `t=0`; if not the case, the initial time is subtracted from all time values.
Second, the in-phase component (the `"Gprime"` element in the output) is proportional to the sinus component, whereas the out-of-phase component (the `"Gprimeprime"` element in the output) is proportional to the cosine component

Value

Vector of two named elements (`"Gprime"` and `"Gprimeprime"`).

Author(s)

Thomas Braschler

`headerSearchPatterns` *headerSearchPatterns*

Description

Standard search patterns to analyze result text files produced by the python library particleShearSimulation

Usage

```
headerSearchPatterns()
```

Value

Vector of standard regular expression search patterns

Author(s)

Thomas Braschler

`linear_regression_p_bootstrap`
linear_regression_p_bootstrap

Description

Uses pre-generated bootstrapping matrices for evaluating relations between variables by linear regression

Usage

```
linear_regression_p_bootstrap(x,y,n_agg=5,na.rm=FALSE)
```

Arguments

`x`	Independent variable for the linear regression; this should be a vector.
`y`	Bootstrapping data. This is a matrix with as many rows as there are elements in `x`, and a possible very large number of columns where the number of columns should be multiple of `n_agg`, otherwise the last columns are ignored such as to achieve a number of columns which is a multiple of `n_agg`.
`n_agg`	Number of columns in `y` that should be used together to generate a dataset to be used for the linear model (i.e. passed to lm).
`na.rm`	In random resampling, it may happen that non-evaluatable samples are generated (all values identical, for example). Although generally rare, in large bootstrapping studies, it may be advantages to pass `na.rm=TRUE` so that these rare samples are not taken into account.

Details

This function assembles datasets for linear regression, calls lm for carrying out the linear regression, and returns statistical data averaged over the different runs. This is typically useful to analyze bootstrapping data, where many data sets are drawn from an original, larger dataset.

Technically, the function aims at regression the data in `y` against the independent variable `x`. For this, datasets are progressively assembled from `x` and `n_agg` columns of `y` at a time. Internally, `x` is repeated `n_agg` times by concatenation, and associated with the `n_agg` columns of `y` assembled into a single vector by concatenation. On this dataset, linear regression (aka lm) is run.

From each run of linear regression, an F statistics and degrees of freedom of the numerator (DF1) and denominator (DF2) as well as an adjusted r squared value is obtained. The function averages these over the number of linear regression runs; the average P-value is calculated using pf from these average statistic values and return as the main value.

Value

Single numerical P-value. In addition, attributes accessible via attr are attached:
"F" for the average F-statistics
"DF1" for the number of degrees of freedom in the numerator
"DF2" for the number of degrees of freedom in the denominator
"adj.r.squared" for the adjusted R-squared value as calculated by summary.lm
"p_shapiro" averaged P-value for normality testing. The averaging here is particular: The P-value returned by shapiro.test is first converted to a Z value by qnorm, and the Z values averaged over the linear regression run. The average Z-value is converted back to an average P-value via pnorm

Author(s)

Thomas Braschler

Examples

```
# Example 1: usage
x=c(0,1,2)

# With a known effect
y=matrix(nrow=length(x),ncol=500,data=x)
y=y+matrix(nrow=dim(y)[1], ncol=dim(y)[2], data=rnorm(dim(y)[1]*dim(y)[2]))
linear_regression_p_bootstrap(x,y,n_agg=5)

# Random effect only
y_control=matrix(nrow=dim(y)[1], ncol=dim(y)[2], data=rnorm(dim(y)[1]*dim(y)[2]))
linear_regression_p_bootstrap(x,y_control,n_agg=5)


# Example 2:
# ================================================================================================================

# Use of a more reality-inspired scenario: The problem of evaluation of outcomes from an averaged curve.
# This is inspired from an Arrhenius-type equation, we use here the temperature at half-maximal reaction rate fo
# assessing catalyst activity (assuming Arrhenius-type saturation at higher temperature). Same scenario as for
# but we will use a transformation to address heteroscedasticity

# T is the known temperature, in Kelvin; T0 the activation energy in temperature equivalents
# Let's assume T0 depends on the addition of some catalyst, this is our x. Let's have the dependency quite extrem
x=c(0,3,6) # Concentrations catalyst
T0=exp(x) # Let's say, this is an exponential dependency
T=c(1,3,10,30,100,300,1000) # Known temperatures
N_per_condition=15
# Theoretical values
theory_reaction_rate = matrix(nrow=length(x),ncol=length(T))
# In reality, we would dispose of measured values with variability rather than theoretical values. Let's say we c
measured=data.frame(T=vector(mode="numeric",length=0),x=vector(mode="numeric",length=0),measured_reaction_r
for(ind_x in 1:length(x))
{
for(ind_T in 1:length(T))
{
theory_reaction_rate[ind_x,ind_T]=50*exp(-T0[ind_x]/T[ind_T])
# Let's say we have normal distribution at log scale
# Generate a random sample with 15 values
measurements_random =exp(log(theory_reaction_rate[ind_x,ind_T])+rnorm(N_per_condition))
```

```
measured=rbind(measured,data.frame(T=T[ind_T],x=x[ind_x],measured_reaction_rate=measurements_random))

}
}
plot(measured_reaction_rate ~ T, measured[measured$x==x[1],],pch=21,col="black",bg="black",ylim=c(-10,150))
for(ind_x in 2:length(x))
{
lines(measured_reaction_rate ~ T, measured[measured$x==x[ind_x],],pch=21,col=palette()[ind_x],bg=palette()[

}

# For the evaluation, let's first evaluate things globally. Let's say we are interested in knowing for each catal

v_max = aggregate ( measured_reaction_rate ~ x, measured[measured$T==max(measured$T),],FUN=median,na.rm=TRUE)
# General aggregation per x and T
v = aggregate(measured_reaction_rate ~ x+T,measured,FUN=median,na.rm=TRUE)
for(ind_x in 1:length(x))
{
lines(measured_reaction_rate ~ T, v[v$x==x[ind_x],],col=palette()[ind_x],type="l")
}


# For convenience, define here a function that estimates the temperatures for half-maximal reaction rate

estimate_half_temperature_at_half_maximal_rate<-function(v)
{
x=sort(unique(v$x))
v_max=v[v$T==max(v$T),]

# For each concentration, to find the temperature where the rate is half-maximal, we go through the temperatures
half_temperature = vector(mode="numeric",length=length(x))
names(half_temperature)=x

for(ind_x in 1:length(x))
{
vx = v[v$x==x[ind_x],]
first_point_above = min(which(vx$measured_reaction_rate>v_max$measured_reaction_rate[v_max$x==x[ind_x]]/2))
if(is.infinite(first_point_above))
{
half_temperature[as.character(x[ind_x])]=max(vx$T)
} else {
if(first_point_above==1)
{
half_temperature[as.character(x[ind_x])]=min(vx$T)
} else {
half_temperature[as.character(x[ind_x])]=approx(vx$measured_reaction_rate[c(first_point_above-1,first_point
vx$T[c(first_point_above-1,first_point_above)],xout=v_max$measured_reaction_rate[v_max$x==x[ind_x]]/2)$y
}
}

}

overall_summary = data.frame(x=x, half_temperature=half_temperature)

return(overall_summary)
```

```
}

overall_summary=estimate_half_temperature_at_half_maximal_rate(v)




dev.new()

# As one can see, it is quite difficult to infer much from this data. Even though the difference in half-maximal t
# three experimental points does not allow to conclude much
plot(half_temperature~x, overall_summary,log="y")
# With the log transformation, the standard linear model is at the limit of significance, it depends on the run
summary(lm(log(half_temperature)~x, overall_summary))


# That's where bootstrapping can offer a reasonable alternative
n_agg = 3 # Let's generate three subsampling plots
N_total = n_agg*100 # 100 blocks
n_to_sample = round(N_per_condition/n_agg) # Nominal coverage of 1

y=matrix(nrow=length(x),ncol=N_total)

for(ind_bootstrap in 1:N_total)
{

measured_subsampled=measured[FALSE,]

for(ind_x in 1:length(x))
{
for(ind_T in 1:length(T))
{
to_subsample = measured[measured$T==T[ind_T] & measured$x==x[ind_x],]
measured_subsampled=rbind(measured_subsampled,to_subsample[sample(dim(to_subsample)[1], n_to_sample),])

}
}

v_subsampled=aggregate(measured_reaction_rate ~ x+T,measured_subsampled,FUN=median)

half_temperature_estimate=estimate_half_temperature_at_half_maximal_rate(v_subsampled)

y[match(half_temperature_estimate$x,x),ind_bootstrap]=half_temperature_estimate$half_temperature

}

# x is now the regressor values (catalyst concentrations) as wanted
# With the log transformation for the y (temperature at half-maximum rate), this generally highly significant
linear_regression_p_bootstrap(x,log(y),n_agg=n_agg)

# Example 3
# Basic settings as in example 2, power and analysis
# ======================================

# as before, to get the temperature estimates at half-maximum rate empirically
```

```
estimate_half_temperature_at_half_maximal_rate<-function(v)
{
x=sort(unique(v$x))
v_max=v[v$T==max(v$T),]

# For each concentration, to find the temperature where the rate is half-maximal, we go through the temperatures
half_temperature = vector(mode="numeric",length=length(x))
names(half_temperature)=x

for(ind_x in 1:length(x))
{
vx = v[v$x==x[ind_x],]
first_point_above = min(which(vx$measured_reaction_rate>v_max$measured_reaction_rate[v_max$x==x[ind_x]]/2))
if(is.infinite(first_point_above))
{
half_temperature[as.character(x[ind_x])]=max(vx$T)
} else {
if(first_point_above==1)
{
half_temperature[as.character(x[ind_x])]=min(vx$T)
} else {
half_temperature[as.character(x[ind_x])]=approx(vx$measured_reaction_rate[c(first_point_above-1,first_point
vx$T[c(first_point_above-1,first_point_above)],xout=v_max$measured_reaction_rate[v_max$x==x[ind_x]]/2)$y
}
}

}

overall_summary = data.frame(x=x, half_temperature=half_temperature)

return(overall_summary)


}



get_p_value_estimate<-function(x,T0,T,N_per_condition)
{

theory_reaction_rate = matrix(nrow=length(x),ncol=length(T))
# In reality, we would dispose of measured values with variability rather than theoretical values. Let's say we c
measured=data.frame(T=vector(mode="numeric",length=0),x=vector(mode="numeric",length=0),measured_reaction_r
for(ind_x in 1:length(x))
{
for(ind_T in 1:length(T))
{
theory_reaction_rate[ind_x,ind_T]=50*exp(-T0[ind_x]/T[ind_T])
# Let's say we have normal distribution at log scale
# Generate a random sample with 15 values
measurements_random =exp(log(theory_reaction_rate[ind_x,ind_T])+rnorm(N_per_condition))
measured=rbind(measured,data.frame(T=T[ind_T],x=x[ind_x],measured_reaction_rate=measurements_random))

}
}
```

```
# For the evaluation, let's first evaluate things globally. Let's say we are interested in knowing for each catal

v_max = aggregate ( measured_reaction_rate ~ x, measured[measured$T==max(measured$T),],FUN=median,na.rm=TRUE)
# General aggregation per x and T
v = aggregate(measured_reaction_rate ~ x+T,measured,FUN=median,na.rm=TRUE)



overall_summary=estimate_half_temperature_at_half_maximal_rate(v)



pvals=vector(length=2, mode="numeric")

names(pvals)=c("lm","lm bootstrapping")

pvals["lm"]=coefficients(summary(lm(log(half_temperature)~x, overall_summary)))["x","Pr(>|t|)"]

n_agg = 3 # Let's generate three subsampling plots
N_total = n_agg*100 # 100 blocks
n_to_sample = round(N_per_condition/n_agg) # Nominal coverage of 1

y=matrix(nrow=length(x),ncol=N_total)

for(ind_bootstrap in 1:N_total)
{

measured_subsampled=measured[FALSE,]

for(ind_x in 1:length(x))
{
for(ind_T in 1:length(T))
{
to_subsample = measured[measured$T==T[ind_T] & measured$x==x[ind_x],]
measured_subsampled=rbind(measured_subsampled,to_subsample[sample(dim(to_subsample)[1], n_to_sample),])

}
}

v_subsampled=aggregate(measured_reaction_rate ~ x+T,measured_subsampled,FUN=median,na.rm=TRUE)

half_temperature_estimate=estimate_half_temperature_at_half_maximal_rate(v_subsampled)

y[match(half_temperature_estimate$x,x),ind_bootstrap]=half_temperature_estimate$half_temperature

}

# x is now the regressor values (catalyst concentrations) as wanted

pvals["lm bootstrapping"]=linear_regression_p_bootstrap(x,log(y),n_agg=n_agg,na.rm=TRUE)

return(pvals)

}


# First, the case with a known effect
```

```
x=c(0,2,4) # Concentrations catalyst
T0=exp(x) # Let's say, this is an exponential dependency
T=c(1,3,10,30,100,300,1000) # Known temperatures
N_per_condition=15
N_simulation=100


pfirst=get_p_value_estimate(x,T0,T,N_per_condition)

p_matrix=matrix(ncol=length(pfirst),nrow=N_simulation)

p_matrix[1,]=pfirst

colnames(p_matrix)=names(pfirst)

for(ind in 2:(dim(p_matrix)[1]))
{
p_matrix[ind,]=get_p_value_estimate(x,T0,T,N_per_condition)

cat(paste("Run ", ind, "\n", sep=""))

}


power=pfirst
for(theApproach in names(power))
{
current_data=p_matrix[,theApproach]
current_data=current_data[!is.na(current_data)]
power[theApproach]=sum(current_data<=0.05)/(length(current_data))
}


# Second, false positives
T0=exp(0*x) # No effect of the catalyst


pfirst=get_p_value_estimate(x,T0,T,N_per_condition)

p_matrix_control=matrix(ncol=length(pfirst),nrow=N_simulation)

p_matrix_control[1,]=pfirst

colnames(p_matrix_control)=names(pfirst)

for(ind in 2:(dim(p_matrix_control)[1]))
{
p_matrix_control[ind,]=get_p_value_estimate(x,T0,T,N_per_condition)

cat(paste("Run ", ind, "\n", sep=""))

}


false_positives=pfirst
for(theApproach in names(false_positives))
{
```

```
current_data=p_matrix_control[,theApproach]
current_data=current_data[!is.na(current_data)]
false_positives[theApproach]=sum(current_data<=0.05)/(length(current_data))
}

for_barplot=matrix(nrow=2,ncol=length(false_positives),data=c(false_positives, power),byrow=TRUE)

rownames(for_barplot)=c("False positives (alpha)", "True positives (1-beta, power)")
colnames(for_barplot)=names(false_positives)

barplot(for_barplot,beside=TRUE,legend=TRUE)
```

`measurement_periods` *measurement_periods*

Description

Identification of different phases of the oscillatory shear experiment

Usage

```
measurement_periods(shear_stress_data,baseline_pre_periods,periods,Frequency)
```

Arguments

`shear_stress_data`	A shear stress data frame. Needs to have at least a column named `"t"`
`baseline_pre_periods`	The number of periods of pre-equilibration before applying shear
`periods`	The number of periods of application of shear
`Frequency`	Frequency of applied oscillatory shear

Value

Vector of as many elements as there are rows in `shear_stress_data`. This function returns a factor with four levels: `"pre"`, `"initiation"`, `"oscillatory_measurement"`, and `"post"`. Entries with `"pre"` signify that the time-point is before initiation of the oscillatory shear; `"initiation"` is initiation (first period if `periods>1`); `"oscillatory_measurement"` indicates the appropriate measurement period; `"post"` indicates re-equilibration after cessation of shear

Author(s)

Thomas Braschler

`read_first_lines` *read_first_lines*

Description

Reads the first lines of a text file

Usage

```
read_first_lines(path,n=100)
```

Arguments

`path`	The path to the file to be read
`n`	The number of lines to be read

Author(s)

Thomas Braschler

`read_general_data_from_file` *read_general_data_from_file*

Description

Reads the general text data from the simulation output data files. This is up to, but without, the stress tensor data.

Usage

```
read_general_data_from_file(path,stress_tensor_data_start_line=-1)
```

Arguments

`path`	The path to the file to be read
`stress_tensor_data_start_line`	The line where the stress tensor data (including the header, but not the title) starts

Author(s)

Thomas Braschler

Examples

```
general_data=read_general_data_from_file(system.file("full_python_simulation_output_example.txt",package="p
general_data[71:78]
```

`read_stress_curve_from_file` *read_stress_curve_from_file*

Description

Reads the primary stress curves stored in a text file located at `path`

Usage

```
read_stress_curve_from_file(path,data_start_line=-1,time_column="t",strain_column="strain",
strain_rate_column="strain_rate",shear_stress_measured_at_surface_column="stress_measured_at_sur
shear_stress_from_internal_stress_tensor_column="shear_stress_internal_stress.tensor")
```

Arguments

`path`	Path to the text file
`data_start_line`	If known, provide the first line to read (the column headers; line numbering is assumed to start with 0 in the file); otherwise, provide -1 and an attempt is made to find the first line to read via stress_data_start_line with default arguments
`time_column`	Header of the time column; by default, this is `"t"`
`strain_column`	Header of the strain column; by default, this is `"strain"`
`strain_rate_column`	Header of the strain rate column; by default, this is `"strain_rate"`
`shear_stress_measured_at_surface_column`	Header of the stress column, as measured by force per unit of surface area at the boundaries where shear force is applied. The stress as measured at the surface most closely emulates what will be measured in a rheometer
`shear_stress_from_internal_stress_tensor_column`	Header of the interal stress column; the internal stress is a volume-averaged shear stress corrected for inertial effects, see eq. 15 in Otsuki et al.

Details

If inertial effects can be neglected, or if correction for them is perfect, the internal stress quantification (column `shear_stress_from_internal_stress_tensor_column`) should match the surface stress (column `shear_stress_measured_at_surface_column`). Large discrepancies between the two indicate that the simulation should is primarily dynamic with imperfect correction rather than quasistatic

Author(s)

Thomas Braschler

References

Otsuki, M. and H. Hayakawa, Discontinuous change of shear modulus for frictional jammed granular materials. Phys Rev E, 2017. 95(6-1): p. 062902.

Examples

```
stress_curve=read_stress_curve_from_file(system.file("full_python_simulation_output_example.txt",package="p
plot(strain ~ t, stress_curve,type="l",xlab="t[s]",ylab="strain [-], stress[kPa]")
lines(stress_curve$t, stress_curve$shear_stress_from_internal_stress_tensor/1000,type="l",col="red")
legend("bottomleft",legend=c("strain","stress"),lty=c(1,1),col=c("black","red"))
```

---

read_stress_curve_from_rda_file

read_stress_curve_from_rda_file

Description

Reads the primary stress curves stored in a rda file located at `path`

Usage

```
read_stress_curve_from_rda_file(path,time_column="t",strain_column="strain",
strain_rate_column="strain_rate",shear_stress_measured_at_surface_column="stress_measured_at_sur
shear_stress_from_internal_stress_tensor_column="shear_stress_internal_stress.tensor")
```

Arguments

`path`	Path to the rda
`time_column`	Header of the time column; by default, this is `"t"`
`strain_column`	Header of the strain column; by default, this is `"strain"`
`strain_rate_column`	Header of the strain rate column; by default, this is `"strain_rate"`
`shear_stress_measured_at_surface_column`	Header of the stress column, as measured by force per unit of surface area at the boundaries where shear force is applied. The stress as measured at the surface most closely emulates what will be measured in a rheometer
`shear_stress_from_internal_stress_tensor_column`	Header of the interal stress column; the internal stress is a volume-averaged shear stress corrected for inertial effects, see eq. 15 in Otsuki et al.

Details

If inertial effects can be neglected, or if correction for them is perfect, the internal stress quantification (column `shear_stress_from_internal_stress_tensor_column`) should match the surface stress (column `shear_stress_measured_at_surface_column`). Large discrepancies between the two indicate that the simulation should is primarily dynamic with imperfect correction rather than quasistatic

Author(s)

Thomas Braschler

References

Otsuki, M. and H. Hayakawa, Discontinuous change of shear modulus for frictional jammed granular materials. Phys Rev E, 2017. 95(6-1): p. 062902.

Examples

```
# This is the same data stored in rda format that is also read from text for read_stress_curve_from_file
path=system.file("split/stress_full_python_simulation_output_example.rda",package="particleShearEvaluation"
stress_curve=read_stress_curve_from_rda_file(path)
plot(strain ~ t, stress_curve,type="l",xlab="t[s]",ylab="strain [-], stress[kPa]")
lines(stress_curve$t, stress_curve$shear_stress_from_internal_stress_tensor/1000,type="l",col="red")
legend("bottomleft",legend=c("strain","stress"),lty=c(1,1),col=c("black","red"))
```

---

read_stress_tensor_data_from_file

read_stress_tensor_data_from_file

Description

Reads the stress tensor data in the file located at `path`

Usage

```
read_stress_tensor_data_from_file(path,data_start_line=-1,time_column="t",strain_column="strain"
strain_rate_column="strain_rate",shear_stress_measured_at_surface_column="stress_measured_at_sur
shear_stress_from_internal_stress_tensor_column="shear_stress_internal_stress.tensor")
```

Arguments

`path`	Path to the text file
`data_start_line`	If known, provide the first line to read (the column headers; line numbering is assumed to start with 0 in the file); otherwise, provide -1 and an attempt is made to find the first line to read via stress_data_start_line with default arguments
`time_column`	Header of the time column; by default, this is `"t"`
`strain_column`	Header of the strain column; by default, this is `"strain"`
`strain_rate_column`	Header of the strain rate column; by default, this is `"strain_rate"`
`shear_stress_measured_at_surface_column`	Header of the stress column, as measured by force per unit of surface area at the boundaries where shear force is applied. The stress as measured at the surface most closely emulates what will be measured in a rheometer
`shear_stress_from_internal_stress_tensor_column`	Header of the interal stress column; the internal stress is a volume-averaged shear stress corrected for inertial effects, see eq. 15 in Otsuki et al.

Details

This function returns a plain dataframe, where the components of the various stress tensor are encoded in separate columns. For their meaning, see the particleShear python package.

Value

data.frame with the same columns as the ones configured in the text file, stress tneosr section

Author(s)

Thomas Braschler

References

Love, A.E. H. A Treatise on the Mathematical Theory of Elasticitiy. Cambridge University Press, 1927.
Weber, J. Recherches concernant les contraintes intergranulaires dans les milieux pulvrulents. Bull. Liaison P. et Ch 20, 1-20, 1966.
Otsuki, M. and Hayakawa, H. Discontinuous change of shear modulus for frictional jammed granular materials. Phys Rev E95, 062902
Nicot, F., Hadda, N., Guessasma, M., Fortin, J. and Millet, O. On the definition of the stress tensor in granular media. Int J Solids Struct, 50, 2508-2517, doi: 10.1016/j.ijsolstr.2013.04.001

Examples

```
path=system.file("full_python_simulation_output_example.txt",package="particleShearEvaluation")
stress_tensor_data=read_stress_tensor_data_from_file(path)
# Some of the stress tensors are symmetrical, some not

# The classical Love-Weber tensors are not:
stress_tensor_asymmetry_Love_Weber=stress_tensor_data$stress_tensor_Love_Weber_01-stress_tensor_data$stress
plot(stress_tensor_data$t,stress_tensor_asymmetry_Love_Weber,type="l", ylim=c(-20,20),main="Asymmetry: Love

dev.new()
# Otsuki et al. indicate a dynamic symmetry correction; in our experience, this does not correct the problem. The
# the asymmetry in the Love-Weber tensor to correct
stress_tensor_asymmetry_Otsuki=stress_tensor_data$stress_tensor_peculiar_acceleration_otsuki_01-stress_tens
plot(stress_tensor_data$t,stress_tensor_asymmetry_Otsuki,type="l", ylim=c(-20,20),main="Asymmetry: Otsuki co
lines(stress_tensor_data$t,stress_tensor_asymmetry_Love_Weber,type="l", col="red")

dev.new()
# Stress tensors calculated uniquely by the externally visible forces are also asymmetric
stress_tensor_asymmetry_external=stress_tensor_data$stress_tensor_from_external_forces_01-stress_tensor_dat
plot(stress_tensor_data$t,stress_tensor_asymmetry_external,type="l", ylim=c(-20,20),main="Asymmetry: Stress

dev.new()
# To symmetrize the stress tensors (Nicot et. al), one can use internal acceleration. Here, accounting done as fo
stress_tensor_asymmetry_linear_acceleration=stress_tensor_data$stress_tensor_linear_acceleration_01-stress_
plot(stress_tensor_data$t,stress_tensor_asymmetry_linear_acceleration,type="l", ylim=c(-20,20),main="Asymme
lines(stress_tensor_data$t,stress_tensor_asymmetry_external,type="l",col="red")

dev.new()
# To symmetrize the stress tensors (Nicot et. al), one can use acceleration stress tensors. The importance is mat
stress_tensor_asymmetry_linear_acceleration=stress_tensor_data$stress_tensor_linear_acceleration_01-stress_
plot(stress_tensor_data$t,stress_tensor_asymmetry_linear_acceleration,type="l", ylim=c(-20,20),main="Asymme
lines(stress_tensor_data$t,stress_tensor_asymmetry_external,type="l",col="red")
legend("topleft",legend=c("linear acceleration correction of total system","external force tensor"),lty=c(1,1

dev.new()
# To symmetrize the stress tensors (Nicot et. al), one can use internal acceleration. As above, but accounting as
stress_tensor_asymmetry_internal_torques=stress_tensor_data$stress_tensor_internal_tangential_torque_01-str
plot(stress_tensor_data$t,stress_tensor_asymmetry_internal_torques,type="l", ylim=c(-20,20),main="Asymmetry
lines(stress_tensor_data$t,stress_tensor_asymmetry_Love_Weber,type="l",col="red")
legend("topleft",legend=c("internal torque correction","Love-Weber"),lty=c(1,1),col=c("black","red"))
```

```
# The relevant shear stress for evaluation is the column stress_tensor_data$shear_stress_internal_stress.tenso
# This is also  -(stress_tensor_data$stress_tensor_internal_tangential_torque_01+stress_tensor_data$stress_t
# -(stress_tensor_data$stress_tensor_internal_tangential_torque_10+stress_tensor_data$stress_tensor_Love_We
plot(stress_tensor_data$t,stress_tensor_data$shear_stress_internal_stress.tensor,type="l",ylab="Shear stres
```

`read_stress_tensor_data_from_rda_file`

read_stress_tensor_data_from_rda_file

Description

Reads the stress tensor data in the file located at `path`

Usage

```
read_stress_tensor_data_from_rda_file(path)
```

Arguments

`path`	Path to the rda file

Details

This merely reads the rda file specified and returns the `stress_data` variable which needs to be contained in the r data file specified by `path`

Value

The `stress_data` variable read from the .rda file indicated by `path`.

Author(s)

Thomas Braschler

References

Love, A. E. H. A Treatise on the Mathematical Theory of Elasticity. (Cambridge University Press, 1927).
Weber, J. Recherches concernant les contraintes intergranulaires dans les milieux pulvurulents. Bull. Liaison P. et Ch.20, 1-20 (1966).
Otsuki, M. & Hayakawa, H. Discontinuous change of shear modulus for frictional jammed granular materials. Phys Rev E95, 062902
Nicot, F., Hadda, N., Guessasma, M., Fortin, J. & Millet, O. On the definition of the stress tensor in granular media. Int J Solids Struct50, 2508-2517, doi:10.1016/j.ijsolstr.2013.04.001 (2013).

See Also

read_stress_tensor_data_from_file for reading the data directly from the large, raw text files; and split_tensor_and_overview_data for splitting the large direct output of the particleShear Python library into a header and rda file to save space

Examples

```
path=system.file("split/stress_full_python_simulation_output_example.rda",package="particleShearEvaluation"
stress_tensor_data=read_stress_tensor_data_from_rda_file(path)

#This should be the same variable as the one read with read_stress_tensor_data_from_file, see that example for f
```

---

split_tensor_and_overview_data

split_tensor_and_overview_data

Description

Splits an output file generated by the python simulation into a header part as a small text file and an rda file for the stress tensor data

Usage

```
split_tensor_and_overview_data(origin_root,path,file_name,destination_root,overwrite=FALSE)
```

Arguments

origin_root	Root path for the full output file
path	Path (folders) within the origin_root location
file_name	File name of the text file to be split
destination_root	Destination path. The function expects the folder structure within the destination location to be the same as in the original location; no folders are created by this function.
overwrite	Should already existing text and rda files be overwritten?

Details

The original text file will be split into a header text file, having the same file name as the original file but preceded by "header_", and a stress tensor rda file. The file name of the rda file is composed of the prefix "stress_", followed by the origina filename, followed by the extension which is ".rda" instead of ".txt". The aim of the splitting operation is permit more rapid access to the stress tensor data, as reading a text file is substantially slower than reading a .rda file

Author(s)

Thomas Braschler

Examples

```
## Not run:
split_tensor_and_overview_data(origin_root=system.file("",package="particleShearEvaluation"),path="","full_

## End(Not run)
```

`stress_data_start_line` *stress_data_start_line*

Description

Finds the line where the stress data starts in the file identified by `path`

Usage

```
stress_data_start_line(path,data_header_pattern="Detailed[a-zA-Z0-9\\s].*data",n_header=200)
```

Arguments

`path`	Path to the text file
`data_header_pattern`	Pattern identifying the line prior to data start (perl-style, see grep).
`n_header`	Number of lines to read while searching for the information. Provide `n_header=-1` for reading the entire file. This argument is passed to read_first_lines, used internally.

Details

The `pattern` argument should uniquely identify the data start line (if not possible, it should at least identify the first occurrence). Also, it should match the text line right above the first data line to read. If in addition to numerical data, there is a header line labelling the data, the pattern needs to identify a text line ONE LINE ABOVE the header line; alternatively, 1 needs to be subtracted from the line number returned

Author(s)

Thomas Braschler

Index

∗ **misc**
evaluate_stress_curve, 2
file_information_table_from_folder, 3
file_information_table_from_folders, 3
find_info_from_file, 5
find_info_from_text, 5
find_infos_from_file, 4
G_from_demodulation, 14
general_linear_regression_p_bootstrap, 6
get_yield_point, 13
headerSearchPatterns, 15
linear_regression_p_bootstrap, 16
measurement_periods, 23
read_first_lines, 24
read_general_data_from_file, 24
read_stress_curve_from_file, 25
read_stress_curve_from_rda_file, 26
read_stress_tensor_data_from_file, 27
read_stress_tensor_data_from_rda_file, 29
split_tensor_and_overview_data, 30
stress_data_start_line, 31
∗ **package**
particleShearEvaluation-package, 2

attr, *17*

coefficients, *6*

data.frame, *27*

evaluate_stress_curve, 2

family, *6*
file_information_table_from_folder, 3
file_information_table_from_folders, 3, *4*
find_info_from_file, 5
find_info_from_text, 5
find_infos_from_file, 4

G_from_demodulation, 14
general_linear_regression_p_bootstrap, 6
get_yield_point, 13
glm, *6*, *7*
grep, *31*

headerSearchPatterns, *3*, 15

linear_regression_p_bootstrap, 16
lines, *15*
lm, *6*, *7*, *16*

measurement_periods, 23

particleShearEvaluation-package, 2
pf, *16*
plot, *15*
plot.counts *(*particleShearEvaluation-package*)*, 2
pnorm, *17*

qnorm, *17*

read_first_lines, 24, *31*
read_general_data_from_file, 24
read_stress_curve_from_file, 25
read_stress_curve_from_rda_file, 26
read_stress_tensor_data_from_file, 27, *29*
read_stress_tensor_data_from_rda_file, 29

shapiro.test, *17*
split_tensor_and_overview_data, *29*, 30
stress_data_start_line, *25*, *27*, 31
summary.glm, *6*
summary.lm, *17*