

Package: tdaR (via r-universe)

September 20, 2026

Type Package

Title Transition Data Analysis

Version 0.1.0

Description An R interface to 'TDA' 6.4, Goetz Rohwer's and Ulrich Poetter's program for transition data analysis, built into the package as a library. Its event history and transition rate models, regression, sequence, cluster, graph and spatial methods and its matrix language are available as R functions that take data frames and return R objects. The package keeps the program and its methods usable and documented, and reproduces results obtained with 'TDA'.

URL <https://github.com/JanMarvin/TDA>, <https://janmarvin.github.io/TDA>,
<https://www.stat.rub.de/tda.html>

BugReports <https://github.com/JanMarvin/TDA/issues>

Copyright see inst/COPYRIGHTS

Language en-GB

Depends R (>= 3.5.0)

Imports stats, utils

Suggests testit, sf, zoo, survival, systemfit, readspss, readstata13,
readxl, foreign, igraph, knitr, rmarkdown, MASS

License GPL (>= 2)

VignetteBuilder knitr

Encoding UTF-8

NeedsCompilation yes

Config/roxygen2/version 8.1.0

Remotes JanMarvin/readspss

Repository <https://janmarvin.r-universe.dev>

Date/Publication 2026-09-20 10:40:11 UTC

RemoteUrl <https://github.com/JanMarvin/TDA>

RemoteRef HEAD

RemoteSha 95e5eea859f5ecefceb35c4d63b4dba1cf155cb4

RemoteSubdir tdaR

Contents

lvl	6
plot.tda_graph	7
plot.tda_ple	8
plot.tda_spatial	10
tda_arcd	11
tda_arcv	12
tda_arcvc	13
tda_bfc	14
tda_boolean_min	14
tda_bounds	16
tda_brr	17
tda_ccnt	18
tda_check_ps	19
tda_clpyr	20
TDA_CLUSTER	20
tda_com	22
tda_combine_ps	22
tda_conj	23
tda_conjoint	24
tda_constrain	25
tda_control	26
tda_cov	28
tda_cutree	29
tda_cwt_norm	30
tda_dblock	31
tda_derive	31
tda_diagnostics	32
tda_dma	33
tda_dmet	35
tda_dmet1	36
tda_dple	36
tda_dstat	39
tda_eigen	40
tda_episodes	41
tda_esort	43
tda_evalfi	45
tda_expressions	46
TDA_FAMILIES	47
tda_fml	51
tda_freg	54
tda_freq	56

tda_frml	57
tda_g	60
tda_g_analyses	61
tda_gap	67
tda_gap_permute	67
tda_gde	68
tda_gdf	69
tda_gdu	70
tda_gfcf	71
tda_ghd	72
tda_gpro	72
tda_gqap	73
tda_graph	74
tda_gtopo	75
tda_help	76
tda_icov	77
tda_idf	78
tda_ilsreg	80
tda_imean	81
tda_imreg	82
tda_independence	83
tda_ineq	84
tda_inpreg	85
tda_integrate	86
tda_interp	88
tda_interval_wages	89
tda_isd	89
tda_isotonic	90
tda_ivar	91
tda_ivar1	92
tda_ivreg	93
tda_llreg	95
tda_locate_line	97
tda_loglin	98
tda_lsreg	100
tda_mag	104
tda_manual_map	105
tda_map	106
tda_mat	108
tda_mcat	109
tda_mcel	110
tda_mcent	111
tda_mch	112
tda_mcross	113
tda_mcvec	114
tda_mdefg	115
tda_mdiag	115
TDA_MDS	116

tda_mdsn	118
tda_mdsr	119
tda_mev	119
tda_mevs	120
tda_midf	121
tda_midf1	122
tda_midf2	123
tda_midf3	124
tda_minvs	125
tda_mkmet	126
tda_mkp	127
tda_mlp	127
tda_mlpi	129
tda_mlrc_design	130
tda_mls	131
tda_mmul	133
tda_mnc	134
tda_mnrow	135
tda_mpbl	136
tda_mpfrit	137
tda_mpinv	138
tda_mpit	139
tda_mple	140
tda_mproc	141
tda_mpz	142
tda_mqap	143
tda_mqp	144
tda_mreg	145
tda_mrsum	146
tda_mscal1	147
tda_msort	148
tda_msqrtd	149
tda_msrow	150
tda_msvd	151
tda_mtrim	152
tda_mwvec	153
tda_ndvar	154
tda_niset	154
tda_npreg	155
tda_options	156
tda_output	157
tda_output_audit	158
tda_payload	159
tda_pyc	159
tda_pdatd	160
tda_pdatr	161
tda_pl	162
tda_pl_arc	166

tda_pl_axis	167
tda_pl_graph	169
tda_pl_hist	170
tda_pl_panel	173
tda_pl_regression	175
tda_pl_scatter	177
tda_plcirc3	178
tda_plglob3	179
tda_plot_ps	180
tda_plotcm	181
tda_plotr	181
tda_plotsp	182
tda_plsurf3	183
tda_polygons	184
tda_ps	185
tda_ps_objects	186
tda_ps3	187
tda_ptree	191
TDA_QRMODELS	192
tda_quant	195
tda_rank_orders	196
tda_rap	197
tda_rates	198
tda_rcorr	198
tda_rcsv	199
tda_rd	200
tda_read_dbf	201
tda_read_ps	202
tda_read_shapefile	203
tda_read_spatial	203
tda_read_spss	206
tda_read_stata	207
tda_read_sys	208
tda_read_table	209
tda_read_ucinet	210
tda_read_xls	211
tda_recode	212
tda_repsel	213
tda_rfit	214
tda_rmod	215
tda_rng	215
tda_rod	217
tda_rplz	217
tda_rrdat	218
tda_rsys	219
tda_run	220
tda_scla	221
tda_sd	222

tda_sd_analyses	223
tda_sd_ps3	225
tda_sddf	229
tda_segr	231
tda_seq_info	232
tda_seqgc	233
tda_seqm	236
tda_seqmd	238
tda_seqpd	241
tda_seqpe	243
tda_seqpm	244
tda_sga	246
tda_sma	246
tda_smd	247
tda_spatial	248
tda_spl	249
tda_split	250
tda_spmmod	252
tda_strings	253
tda_subm	254
tda_survivor	255
tda_tnet	256
tda_transitions	256
tda_triang	257
tda_ucl	258
tda_unf	259
tda_write_data	260
tda_write_foreign	263
tda_zreg	264
unzoo	266
Index	269

lvl

*A variable's value by category, for tda_qreg***Description**

Marks a set of columns, used inside a [tda_qreg](#) formula, as the same underlying variable's value at each category of the response – price for choice 1, choice 2, choice 3, say, one coefficient shared across all of them – rather than the formula's ordinary one-column-per-predictor, shared-across-categories shape. TDA's "(Z1,Z2,...)" grouped variable-list syntax (what its source calls "Z variables" or "generic variables"), reached through R's `Surv()`-style formula convention (from the survival package – not a dependency of this one, just the same idea) instead of a separate argument.

Usage

```
lvl(...)
```

Arguments

... one column per category of the response, in category order.

Details

lvl() is only meaningful inside a tda_qreg formula – it is not a real transformation, just a marker tda_qreg looks for and removes before fitting, the same way Surv() is not itself computed by coxph(), only recognised by it.

Value

A matrix, for `model.matrix`'s benefit – not meaningful on its own.

Examples

```
# Daganzo's classic mode-choice example (examples/exam/qr4.cf in TDA's
# own sources): commute time by car (Z1), bus (Z2), and train (Z3), one
# coefficient for time itself, shared across all three modes
# (seeded synthetic version: choices generated from utility
# -0.3 * time + Gumbel noise, so the estimate recovers about -0.3)
set.seed(21)
n <- 80
d <- data.frame(Z1 = runif(n, 5, 30), Z2 = runif(n, 5, 30),
                Z3 = runif(n, 5, 30))
u <- -0.3 * as.matrix(d) + matrix(-log(-log(runif(3 * n))), n)
d$Y <- max.col(u)
tda_qreg(Y ~ lvl(Z1, Z2, Z3), d, model = "multinomial_logit",
        nq = 3, intercept = FALSE)$estimates
```

plot.tda_graph

Draw a graph

Description

TDA's graph-drawing command, `pltree`, only draws trees. For anything else – a cycle, more than one component, or a directed graph – this lays the nodes out on a circle and draws it with `tda_pl_graph` instead, which takes explicit coordinates and so can draw any graph.

Usage

```
## S3 method for class 'tda_graph'
plot(
  x,
  width = 110,
```

```

height = 80,
file = "graph.ps",
newpage = TRUE,
layout = c("auto", "circle", "tree"),
...
)

```

Arguments

x	a tda_graph .
width, height	size of the plotting area in millimetres.
file	name for the PostScript file.
newpage	start a new page before drawing.
layout	"auto" (the default) uses <code>pltree</code> when the graph is actually a tree (connected, one fewer edge than nodes) and a circular layout otherwise; "circle" always uses the circular layout, "tree" always tries <code>pltree</code> and lets TDA refuse a non-tree graph with its own error.
...	further options for <code>pltree</code> (layout = "tree") or tda_pl_graph (layout = "circle").

Value

The parsed drawing operations, invisibly.

See Also

Other graph analysis: [tda_dmet\(\)](#), [tda_g\(\)](#), [tda_g_analyses](#), [tda_graph\(\)](#), [tda_locate_line\(\)](#), [tda_ptree\(\)](#), [tda_ucl\(\)](#)

Examples

```

e <- data.frame(from = c(1, 1, 2), to = c(2, 3, 4), value = 1)
g <- tda_graph(e, directed = FALSE) # 4 nodes, 3 edges: already a tree
pdf(NULL); plot(g); dev.off()

# a graph with a cycle is not a tree, and this one is directed besides --
# both are fine with the default circular layout
e2 <- data.frame(from = c(1, 1, 2, 2, 3, 4), to = c(2, 3, 3, 4, 4, 5))
g2 <- tda_graph(e2, directed = TRUE)
pdf(NULL); plot(g2); dev.off()

```

plot.tda_ple

Plot survivor curves and fitted rates

Description

Base graphics, drawn from the tables the estimators already return, so no plotting dependency is needed. Censored times are marked, a band is drawn where a standard error is available, and groups get a legend.

Usage

```
## S3 method for class 'tda_ple'
plot(
  x,
  conf.int = TRUE,
  mark.censored = TRUE,
  col = NULL,
  lty = 1,
  lwd = 1.5,
  xlab = "Time",
  ylab = "Survivor function",
  main = NULL,
  legend = TRUE,
  ylim = c(0, 1),
  ...
)

## S3 method for class 'tda_ltb'
plot(x, ...)

## S3 method for class 'tda_rate'
plot(
  x,
  what = c("survivor", "rate", "density"),
  xlab = "Time",
  ylab = NULL,
  col = "black",
  lwd = 1.5,
  main = NULL,
  ...
)
```

Arguments

<code>x</code>	an estimate from tda_km , <code>tda_ple</code> , <code>tda_ltb</code> , or a fit from tda_rate .
<code>conf.int</code>	draw a band at plus or minus 1.96 standard errors.
<code>mark.censored</code>	mark censoring times with a cross.
<code>col</code> , <code>lty</code> , <code>lwd</code>	colour, line type and width, recycled over groups.
<code>xlab</code> , <code>ylab</code> , <code>main</code> , <code>ylim</code>	as usual.
<code>legend</code>	draw a legend when there is more than one group.
<code>...</code>	passed to <code>plot</code> .
<code>what</code>	for a fitted rate model, which of the survivor, rate or density curves to draw. The fit must have been made with <code>prate</code> .

Value

The object, invisibly.

See Also

Other plotting: [tda_check_ps\(\)](#), [tda_pl\(\)](#), [tda_pl_arc\(\)](#), [tda_pl_axis\(\)](#), [tda_pl_graph\(\)](#), [tda_pl_hist](#), [tda_pl_panel\(\)](#), [tda_pl_regression\(\)](#), [tda_pl_scatter\(\)](#), [tda_plot_ps\(\)](#), [tda_ps\(\)](#), [tda_ps3\(\)](#), [tda_read_ps\(\)](#)

Examples

```
set.seed(40)
d <- data.frame(t = round(rexp(60, 0.1), 1) + 0.5, s = rbinom(60, 1, 0.8))
km <- tda_km(Surv(t, s) ~ 1, d)
plot(km)
```

plot.tda_spatial	<i>Draw spatial data</i>
------------------	--------------------------

Description

sdplot needs a PostScript coordinate system as well as the spatial data, so the ranges have to be given.

Usage

```
## S3 method for class 'tda_spatial'
plot(
  x,
  xlim = NULL,
  ylim = NULL,
  width = 100,
  height = NULL,
  newpage = TRUE,
  what = c("objects", "hull", "hull_per_polygon"),
  symbol = NULL,
  size = NULL,
  lty = NULL,
  lw = NULL,
  gray = NULL,
  ...
)
```

Arguments

<code>x</code>	a tda_spatial .
<code>xlim, ylim</code>	ranges of the coordinate system.
<code>width, height</code>	size of the plotting area in millimetres; the height follows the data's aspect ratio unless given.
<code>newpage</code>	start a new page before drawing.
<code>what</code>	what to draw: "objects" (default), the convex hull over every point, or the convex hull of each polygon separately; <code>sdplot</code> 's <code>opt=</code> .
<code>symbol, size, lty, lw, gray</code>	marker symbol and size, line type, line width in mm, and grey level from 0 (black) to 1 (white); <code>sdplot</code> 's own <code>s=/fs=/lt=/lw=/gs=</code> .
<code>...</code>	further options for <code>sdplot</code> .

Details

Points need a marker or nothing is drawn: give `symbol` (or another TDA symbol number). This is the same trap noted for [tda_sd_plot3](#) – without a marker or a `zvar` attribute, `sdplot` draws nothing at all rather than erroring.

Value

The parsed drawing operations, invisibly.

See Also

Other spatial analysis: [tda_map\(\)](#), [tda_polygons\(\)](#), [tda_read_dbf\(\)](#), [tda_read_shapefile\(\)](#), [tda_read_spatial](#), [tda_sd\(\)](#), [tda_sd_analyses](#), [tda_sd_ps3\(\)](#), [tda_spatial\(\)](#)

Examples

```
d <- data.frame(id = 1:4, x = c(0, 1, 1, 0), y = c(0, 0, 1, 1))
s <- tda_spatial(d)
plot(s, symbol = 1) # a marker is needed; points draw nothing without one
```

 `tda_arcd` |

 Open or close a TDA data archive |

Description

TDA's `arcd`: opens access described by an archive description file, or closes it with `file = NULL`.

Usage

```
tda_arcd(file = NULL, ...)
```

Arguments

file archive description file, or NULL to close.
 ... passed to [tda_run](#).

Value

the printed output, invisibly.

Examples

```
# an archive description file names the zoo archive and then one
# line per member: number, name, type (1 data, 2 variable
# descriptions), record length, records, variables
dr <- tempfile("tda"); dir.create(dr)
zoo <- system.file("extdata", "tda.zoo", package = "tdaR")
file.copy(zoo, file.path(dr, "tda.zoo"))
writeLines(c("tda.zoo", "1 adata.dat 1 24 20 3", "2 avar.dat 2 40 3 0"),
           file.path(dr, "arc.ad"))
out <- tda_arcd("arc.ad", dir = dr)
cat(grep("archive|Checking|adata", out, value = TRUE), sep = "\n")
```

tda_arcv

Print archive variables

Description

TDA's arcv on an open archive. It requires an open archive (see [tda_arcd](#)) containing a type-2 (variable description) member.

Usage

```
tda_arcv(out, data_file = NULL, archive = NULL, ...)
```

Arguments

out output file name.
 data_file optional data-file filter.
 archive the archive description (.zad) to load first. arcv reads an archive that arcd has loaded, and TDA keeps no state between runs, so this is needed unless the caller issues arcd in the same run themselves.
 ... passed to [tda_run](#).

Value

the printed output, invisibly.

Examples

```
d <- tempfile(); dir.create(d)
for (f in c("tda.zad", "tda.zoo"))
  file.copy(system.file("extdata", f, package = "tdaR"), d)
tda_arvcv("vars.vd", archive = "tda.zad", dir = d)
readLines(file.path(d, "vars.vd"))
```

tda_arvcv

*Check a variable description file***Description**

TDA's arvcv. It requires an open archive (see [tda_arcd](#)) containing a type-2 (variable description) member.

Usage

```
tda_arvcv(file, rewrite = NULL, suffix = c("number", "lower", "upper"), ...)
```

Arguments

file	the description file.
rewrite	optional path for a rewritten file with unique names.
suffix	how names are made unique: "number", "lower", or "upper" (file number or file name in either case).
...	passed to tda_run .

Value

the printed output, invisibly.

Examples

```
# the file arcv writes, checked for duplicate variable names
d <- tempfile(); dir.create(d)
for (f in c("tda.zad", "tda.zoo"))
  file.copy(system.file("extdata", f, package = "tdaR"), d)
tda_arvcv("vars.vd", archive = "tda.zad", dir = d)
# arcv writes a template with every definition commented out, for the
# caller to uncomment; arcv reports its verdict on it either way
out <- tda_arvcv("vars.vd", dir = d)
cat(grep("variable|once", out, value = TRUE), sep = "\n")
```

tda_bfc	<i>Context dependencies in Boolean functions</i>
---------	--

Description

TDA's bfc: Y as a Boolean function of binary X's.

Usage

```
tda_bfc(y, x, undefined = c("dontcare", "as_one", "as_zero"), ...)
```

Arguments

- y binary outcome.
- x binary matrix.
- undefined how undefined arguments count: "dontcare", "as_one", or "as_zero".
- ... passed to [tda_run](#).

Value

the printed output (implicants).

Examples

```
x <- expand.grid(a = 0:1, b = 0:1)
r <- tda_bfc(y = as.integer(x$a & x$b), x = x)
cat(head(tda_payload(r), 12), sep = "\n")
```

tda_boolean_min	<i>Boolean function minimization</i>
-----------------	--------------------------------------

Description

Which combinations of binary conditions produce the outcome? TDA's bfa builds the truth table of y over the condition variables, finds all prime implicants (Quine-McCluskey), and selects minimal covers. This is the machinery behind crisp-set QCA-style analyses.

Usage

```
tda_boolean_min(
  y,
  conditions,
  data = NULL,
  undefined = c("dont_care", "true", "false"),
  algorithm = c("lawler", "petrick", "lawler2", "none"),
  max_implicants = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

<code>y</code>	the outcome: a 0/1 vector, or a column name in data.
<code>conditions</code>	the condition variables: a 0/1 matrix or data frame, or column names in data (at most 15).
<code>data</code>	optional data frame supplying the columns.
<code>undefined</code>	how rows never observed are treated: "dont_care" (default), "true", or "false".
<code>algorithm</code>	cover selection: "lawler" (default), "petrick", "lawler2", or "none" for prime implicants only.
<code>max_implicants</code>	storage cap.
<code>options</code>	a named list of further TDA options for bfa (ptab, prot, ...).
<code>dir</code>	working directory.
<code>...</code>	passed to tda_run .

Value

A list: `$selections`, a data frame with one row per term of each minimal cover – selection number and one column per condition holding 1, 0, or NA for "does not matter" – and `$expression`, the printed symbolic forms.

See Also

Other optimization: [tda_mlp\(\)](#), [tda_mlpi\(\)](#), [tda_mls\(\)](#), [tda_mqap\(\)](#), [tda_mqp\(\)](#)

Examples

```
d <- expand.grid(X1 = 0:1, X2 = 0:1, X3 = 0:1)
d$Y <- as.integer(d$X1 & d$X2 | d$X3)
tda_boolean_min("Y", c("X1", "X2", "X3"), data = d)$selections
```

tda_bounds

*Bounds from an interval regression***Description**

An interval regression identifies its coefficient only up to an interval, so `coef()` would be misleading. This returns the bounds TDA reports.

Usage

```
tda_bounds(x, what = c("beta", "alpha", "mean_y", "mean_x", "var_x"))
```

Arguments

<code>x</code>	a fit from <code>tda_ivreg</code> and friends.
<code>what</code>	which bounds: the slope, the intercept (<code>tda_ilsreg</code> only), or the mean or variance summaries.

Value

A named vector with lower and upper, or NULL.

See Also

Other interval-valued data: `tda_idf()`, `tda_ilsreg()`, `tda_imean()`, `tda_imreg()`, `tda_inpreg()`, `tda_ivar1()`, `tda_ivreg()`, `tda_sddf()`

Examples

```
set.seed(37)
n <- 40
xlo <- round(rnorm(n), 2); xhi <- xlo + round(runif(n, 0.5, 1.5), 2)
ylo <- round(1 + 0.6 * xlo + rnorm(n, sd = 0.3), 2)
yhi <- ylo + round(runif(n, 0.5, 1.5), 2)
d <- data.frame(xlo = xlo, xhi = xhi, ylo = ylo, yhi = yhi)
ivf <- tda_ivreg(iv(ylo, yhi) ~ iv(xlo, xhi), d)
tda_bounds(ivf, "beta")
tda_bounds(ivf, "mean_x")

# ilsreg: an interval response, a point regressor -- beta and alpha (the
# intercept) both come back bounded, not just the slope
set.seed(2)
d2 <- data.frame(x = round(rnorm(30), 2))
d2$ylo <- 2 + 0.5 * d2$x + rnorm(30, sd = 0.2)
d2$yhi <- d2$ylo + round(runif(30, 0.3, 1), 2)
lsf <- tda_ilsreg(iv(ylo, yhi) ~ x, d2)
tda_bounds(lsf, "beta")
tda_bounds(lsf, "alpha")
```

tda_brr

*Balanced repeated replication design***Description**

`brr`: builds the replicate-weight matrix balanced repeated replication needs – a variance-estimation technique from survey statistics that works by refitting an estimator on `nr` half-sample replicates of the data (`nr`, the number of replications required, is determined from `strata` and `secu` and reported back) and taking the spread across them as the variance, rather than a model-based formula. This returns the design itself – TDA’s orthogonal (Hadamard-based) construction, not practical to reproduce independently in R – so it can be applied to whatever estimator is being replicated; actually refitting a model `nr` times and computing the BRR variance from the results is ordinary R code once the design is in hand, not shown here since it depends entirely on what is being estimated.

Usage

```
tda_brr(strata, secu, dir = tempfile("tda"), ...)
```

Arguments

<code>strata</code>	number of strata.
<code>secu</code>	number of sampling units per stratum; 2 in the overwhelming majority of real BRR designs, and the only value tested here.
<code>dir</code>	working directory.
<code>...</code>	passed to <code>tda_run</code> .

Details

Not every (`strata`, `secu`) combination succeeds: `secu` must be at least 2 (one stratum, one unit gives nothing to replicate), and at least one combination that satisfies every other documented constraint (`brr = 4, 2`) can still fail with “no success in calculating orthogonal weights”: 2, 3, and 8 strata at `secu = 2` all succeed, 4 does not. This is a limitation in TDA’s Galois-field construction for specific sizes, not something this wrapper can lift.

Value

A list: `replications` (`nr`, how many are needed), and `design`, the strata by replications matrix itself (TDA’s description: `design[i, j]` is which sampling unit of stratum `i` to use in replication `j`).

See Also

Other descriptive statistics: `tda_cov()`, `tda_dstat()`, `tda_freq()`, `tda_independence()`, `tda_ineq()`, `tda_loglin()`, `tda_quant()`, `tda_rcorr()`, `tda_segr()`, `tda_subm()`

Examples

```
b <- tda_brr(3, 2)
b$replications
b$design
```

tda_ccnt

*Inspect a raw file***Description**

Small file-diagnostic utilities from TDA's toolkit, from an era before assuming `wc`, `od` or `split` were available: `tda_ccnt` a character frequency table, `tda_lcnt` a record-length frequency table, `tda_dump` a hex dump, and `tda_dsplrit` splits a file into fixed-size parts on disk.

Usage

```
tda_ccnt(file, dir = tempfile("tda"), ...)

tda_lcnt(file, noc = NULL, dir = tempfile("tda"), ...)

tda_dump(file, nc = NULL, offset = NULL, dir = tempfile("tda"), ...)

tda_dsplrit(file, len = NULL, dir = tempfile("tda"), ...)
```

Arguments

<code>file</code>	path to the file to inspect.
<code>dir</code>	working directory; for <code>tda_dsplrit</code> , also where the parts are written, alongside the original file.
<code>...</code>	passed to tda_run .
<code>noc</code>	for <code>tda_lcnt</code> , the maximum number of records to read.
<code>nc</code>	for <code>tda_dump</code> , how many bytes to show; all of them by default.
<code>offset</code>	for <code>tda_dump</code> , where to start, in bytes.
<code>len</code>	for <code>tda_dsplrit</code> , the size of each part in bytes.

Value

`tda_ccnt` and `tda_lcnt` return a data frame. `tda_dump` returns the hex dump as a character vector, one line each. `tda_dsplrit` returns the paths of the parts it wrote.

See Also

Other TDA infrastructure: [tda_help\(\)](#), [tda_output\(\)](#), [tda_read_table\(\)](#), [tda_run\(\)](#), [tda_write_data\(\)](#)

Examples

```
f <- tempfile()
writeLines(c("hello world", "foo bar baz", "x"), f)
tda_ccnt(f)
tda_lcnt(f)
cat(tda_dump(f), sep = "\n")

f2 <- tempfile()
writeLines(rep("0123456789", 10), f2)
tda_dsplitt(f2, len = 20)
```

tda_check_ps

*Audit a parse against the PostScript it came from***Description**

Counts the drawing operators in the file and the operations the parser produced, and reports anything unaccounted for. Reading a plot back is easy to get half right – an operator that is silently ignored looks exactly like one that is not there – so this compares the two directly rather than leaving it to the eye.

Usage

```
tda_check_ps(file, ...)
```

Arguments

`file` a .ps file, a TDA run, or a parse from [tda_read_ps](#).
`...` passed to [tda_read_ps](#).

Value

A data frame with one row per operator: how often it appears in the file, whether the parser handles it, and how many operations resulted.

See Also

Other plotting: [plot.tda_ple\(\)](#), [tda_pl\(\)](#), [tda_pl_arc\(\)](#), [tda_pl_axis\(\)](#), [tda_pl_graph\(\)](#), [tda_pl_hist](#), [tda_pl_panel\(\)](#), [tda_pl_regression\(\)](#), [tda_pl_scatter\(\)](#), [tda_plot_ps\(\)](#), [tda_ps\(\)](#), [tda_ps3\(\)](#), [tda_read_ps\(\)](#)

Examples

```
p <- tda_ps(xlim = c(0, 10), ylim = c(0, 10))
p <- tda_pl_circle(p, at = c(5, 5), r = 2)
tda_check_ps(tda_ps_file(p))
```

tda_clpyr	<i>Pyramidal clustering</i>
-----------	-----------------------------

Description

TDA's clpyr on a distance matrix.

Usage

```
tda_clpyr(d, linkage = c("single", "complete"), ...)
```

Arguments

d	symmetric distance matrix.
linkage	"single" or "complete" link.
...	passed to tda_run .

Value

the printed output.

Examples

```
r <- tda_clpyr(as.matrix(dist(c(0, 1, 5, 6))))
# the top of the pyramid joins everything: 4 objects spanning 1..4
tail(r$pyramid, 1)
```

TDA_CLUSTER	<i>Clustering from a dissimilarity matrix</i>
-------------	---

Description

TDA's clustering commands, all of which take a dissimilarity and so accept a dist – including the one [tda_seqm](#) returns.

Usage

```
TDA_CLUSTER

tda_cluster(
  d,
  method = "hierarchical",
  algorithm = c("centers", "diameter"),
  min_size = NULL,
  max_splits = NULL,
```

```

    options = list(),
    dir = tempfile("tda"),
    ...
)

```

Arguments

<code>d</code>	a dist, or a square dissimilarity matrix.
<code>method</code>	one of the names in TDA_CLUSTER, matched partially.
<code>algorithm</code>	for <code>method = "hierarchical"</code> only: split by "centers" (default, maximally different cluster centers) or "diameter" (minimal-diameter partitions) – <code>hcld</code> 's <code>alg=</code> .
<code>min_size</code>	for <code>algorithm = "centers"</code> , the minimum cluster size (<code>hcld</code> 's <code>min=</code>).
<code>max_splits</code>	for <code>algorithm = "diameter"</code> , the maximum number of splits (<code>hcld</code> 's <code>max=</code>).
<code>options</code>	a named list of further TDA options, passed through.
<code>dir</code>	working directory.
<code>...</code>	passed to <code>tda_run</code> .

Details

The default, "hierarchical" (`hcld`), writes a ragged table – a node, its member count, then that many member indices – which is not a rectangular `data.frame`, so it is parsed separately into clusters, a named list of member-index vectors, one per node of the divisive tree (the root first, then each split). "unidimensional" and "additive" report their result only as text (a final partition or a fitted tree), never through `df=`; read `$run$output` for those.

Value

An object carrying a table, or for "hierarchical" clusters instead (see Details). For "hierarchical" the object also carries merges: one row per split, the two clusters, their sizes and their diameters.

See Also

Other clustering: `TDA_MDS`, `tda_conjoint()`, `tda_cutree()`, `tda_dma()`, `tda_pdatd()`

Examples

```

d <- dist(matrix(c(1, 2, 3, 8, 9, 10), ncol = 1))
tda_cluster(d)$clusters # the divisive tree: root, then each split

```

tda_com	<i>Combinatorial patterns</i>
---------	-------------------------------

Description

TDA's com: enumerate tuples, sets, permutations or partitions into an output file.

Usage

```
tda_com(
  pattern = c("tuples", "subsets", "m_tuples", "permutations", "partitions",
    "partitions_m", "bipartitions"),
  n = 1,
  m = 1,
  ...
)
```

Arguments

pattern	one of "tuples", "subsets", "m_tuples", "permutations", "partitions", "partitions_m", "bipartitions".
n, m	dimensions (n the base size, m where the pattern needs it).
...	passed to tda_run .

Value

the output file's lines.

Examples

```
# the six 2-element subsets of {0, 1, 2, 3}: a row number, then the members
tda_com("subsets", n = 4, m = 2)
# the five partitions of 4: 4, 3+1, 2+2, 2+1+1, 1+1+1+1
tda_com("partitions", n = 4)
```

tda_combine_ps	<i>Combine several PostScript plots into one file</i>
----------------	---

Description

tda_combine_ps reaches TDA's dplot command, which lays out a grid of previously-created PostScript files (each from its own tda_ps() session, already run) into a single output file – one row per element of rows, side by side within a row. Unlike every other tda_pl_* function here, this is not a command added to an in-progress session: dplot combines files that already exist on disk into a new one of its own, so this takes file paths (or tda_ps objects that have already been run, via [tda_ps_file](#)) and runs standalone, the way [tda_run](#) does.

Usage

```
tda_combine_ps(
  rows,
  width = 120,
  height = 80,
  origin = c(10, 10),
  file = "combined.ps",
  ...
)
```

Arguments

<code>rows</code>	A list of PostScript files to lay out in a grid, one row per list element; each element is a character vector of file paths (or <code>tda_ps</code> run objects) placed side by side in that row.
<code>width, height</code>	Size of the combined output, in mm (TDA's defaults: 120 and 80).
<code>origin</code>	Physical origin of the combined plot, <code>c(x, y)</code> in mm (TDA's default: <code>c(10, 10)</code>).
<code>file</code>	Name of the combined output file.
<code>...</code>	Further options passed to <code>tda_run</code> .

Value

The result of `tda_run`, whose own run directory holds file.

Examples

```
# two small plots, then both side by side in one file
p1 <- tda_ps(xlim = c(0, 10), ylim = c(-1, 1), width = 55, height = 40)
p1 <- tda_pl_axes(tda_pl_function(p1, "sin(x)"))
p2 <- tda_ps(xlim = c(0, 10), ylim = c(0, 100), width = 55, height = 40)
p2 <- tda_pl_axes(tda_pl_function(p2, "x * x"))
r <- tda_combine_ps(list(c(tda_ps_file(p1), tda_ps_file(p2))))
file.exists(file.path(r$dir, "combined.ps"))
```

tda_conj

*Conjoint analysis***Description**

TDA's `conj` (alternating least squares) on a preference variable and factor columns.

Usage

```
tda_conj(y, x, algorithm = c("monotone", "lp"), ...)
```

Arguments

`y` preference values.
`x` factor matrix.
`algorithm` "monotone" regression or "lp" (linear programming).
`...` passed to `tda_run`.

Value

the printed output.

Examples

```

x <- expand.grid(a = 1:2, b = 1:2)
r <- tda_conj(y = c(4, 3, 2, 1), x = x)
cat(tail(tda_payload(r), 8), sep = "\n") # the part-worths

```

tda_conjoint	<i>Non-metric conjoint analysis</i>
--------------	-------------------------------------

Description

nmca is conjoint measurement: it takes a rank order over profiles and the categorical factors describing them, and estimates the part-worth of each factor level. It draws a Hasse diagram of the order as part of the output.

Usage

```
tda_conjoint(formula, data, options = list(), dir = tempfile("tda"), ...)
```

Arguments

`formula` rank ~ f1 + f2, where the response is the rank order and the right-hand side names the factors.
`data` a data frame. Factor levels are numbered from 1 upwards, and a value outside that range is rejected by TDA with the record it occurred in.
`options` a named list of further TDA options, passed through.
`dir` working directory.
`...` passed to `tda_run`.

Value

An object carrying the run; the diagram and the estimates are in `$run$output`.

See Also

Other clustering: `TDA_CLUSTER`, `TDA_MDS`, `tda_cutree()`, `tda_dma()`, `tda_pdatd()`

Examples

```
d <- expand.grid(price = 1:2, brand = 1:2)
d$rank <- c(4, 2, 3, 1)
cj <- tda_conjoint(rank ~ price + brand, d)
cat(tda_payload(cj$run), sep = "\n") # the nmca section itself
```

tda_constrain

*Build a linear parameter constraint***Description**

[tda_rate](#) takes constraints in TDA's notation, where parameters are numbered in the order they are estimated: "b3 - b10 = 0" makes the third and tenth equal. This builds such a string from a fitted model's coefficient names, so the numbering does not have to be counted out by hand.

Usage

```
tda_constrain(fit, ...)
```

Arguments

<code>fit</code>	a fit from tda_rate , used for its coefficient names.
<code>...</code>	constraints, each a character vector of two coefficient names to be held equal, or a formula like <code>EDU ~ PRES</code> .

Value

A character vector of constraints for `tda_rate`.

See Also

Other rate models: [tda_control\(\)](#), [tda_dple\(\)](#), [tda_rates\(\)](#), [tda_split\(\)](#), [tda_survivor\(\)](#), [tda_transitions\(\)](#), [vcov.tda_fit\(\)](#)

Examples

```
d <- tda_rrdat(states = 4)
f <- tda_rate(Surv(TFP, DES) ~ EDU + PRES, d)
tda_constrain(f, c("0->1: EDU", "0->2: EDU"))
```

tda_control	<i>Convergence control</i>
-------------	----------------------------

Description

Convergence control

Usage

```
tda_control(
  algorithm = NULL,
  tolsq = NULL,
  tolsq = NULL,
  tolg = NULL,
  tolf = NULL,
  maxit = NULL,
  tol_param = NULL,
  tol_var = NULL,
  tol_reduction = NULL,
  step_length = NULL,
  step_reduction = NULL,
  step_min = NULL,
  maxit_line = NULL,
  criterion = NULL,
  derivatives = NULL
)

tda_strict(tol = 1e-12)

tda_converged(x)
```

Arguments

algorithm	TDA minimiser number, its mina=: 1 direct search, 2 simplex, 3 conjugate gradients, 4 BFGS, 5 Newton (I, TDA's own default), 6 Newton (II), 7 CES with a quadratic model, 8 CES with a tensor model. 7 and 8 use the scale-invariant criterion.
tolsg, tolsq	tolerances for the scaled gradient and scaled parameter change, used by algorithms 7 and 8.
tolg, tolf	tolerances for the gradient norm and the function value, used by the default algorithm.
maxit	iteration cap, TDA's mxit=.
tol_param, tol_var, tol_reduction	three more tolerances TDA's minimiser reads for other algorithms/situations – tolp= (parameter change, direct search and others), tolv= (Simplex), tols= (reduction factor, direct search).

step_length, step_reduction, step_min	step-size controls for TDA's line search – slen=, sred=, smin=.
maxit_line	cap on line-search iterations within one step – TDA's mxitl=, default 50.
criterion	which convergence test to apply – TDA's crit=, 1 to 3. Confirmed limited: it cannot select the scale-invariant Dennis-Schnabel test described under Criteria above, which is reachable only through algorithm = 7 or 8 directly, not through this argument.
derivatives	0 for analytical derivatives (TDA's default when unset), 1 or 2 for numerical approximations of varying cost – TDA's dopt=. Applies only to algorithm = 7 or 8; harmless but inert for every other algorithm.
tol	tolerance for tda_strict, applied to both tolsg and tolspl.
x	a fitted model.

Value

A list of options for control=; tda_converged returns TRUE, FALSE, or NA where TDA reports nothing.

Criteria

TDA's default minimiser stops on the norm of the gradient against an absolute tolerance, plus a loose disjunction that also stops when the change in function value is small. Algorithms 7 and 8 switch to optstp(), the Dennis-Schnabel test on the largest *relative* gradient and relative parameter step. That is scale-invariant, closer to what R's optimisers do, and gives results that reproduce across platforms. It is reachable only through algorithm: TDA's crit= accepts 1 to 3 and cannot select it.

The default is TDA's, so a plain call reproduces a plain command file. tda_strict() opts in.

A caution

Algorithms 7 and 8 also lower the iteration cap to 20 when derivatives are numerical, against 100 otherwise, so asking for a tighter tolerance can turn a converged fit into a truncated one. Every fit reports how it stopped, warns if it did not converge, and tda_converged() reads that back.

See Also

Other rate models: [tda_constrain\(\)](#), [tda_dple\(\)](#), [tda_rates\(\)](#), [tda_split\(\)](#), [tda_survivor\(\)](#), [tda_transitions\(\)](#), [vcov.tda_fit\(\)](#)

Examples

```
set.seed(36)
d <- data.frame(x = rnorm(200))
d$y <- rbinom(200, 1, plogis(0.3 + 0.8 * d$x))
f <- tda_qlreg(y ~ x, d, control = tda_control(maxit = 50, tolg = 1e-8))
coef(f)
tda_converged(f)

# tda_strict: the same idea, one shared tolerance for both algorithm 7/8
```

```
# criteria at once
f2 <- tda_qlreg(y ~ x, d, control = tda_strict())
coef(f2)
```

tda_cov

Covariance matrix

Description

The $n - 1$ -denominator covariance matrix, as `stats::cov()` computes it, extended to case weights when a `cwt` command is active. [tda_corr](#) is its correlation counterpart.

Usage

```
tda_cov(..., options = list(), dir = tempfile("tda"))
```

Arguments

<code>...</code>	a data frame, a matrix, or vectors.
<code>options</code>	a named list of further TDA options, passed through.
<code>dir</code>	working directory.

Value

An object carrying `matrix`, the covariance matrix with the variables' names on both dimensions.

See Also

Other descriptive statistics: [tda_brr\(\)](#), [tda_dstat\(\)](#), [tda_freq\(\)](#), [tda_independence\(\)](#), [tda_ineq\(\)](#), [tda_loglin\(\)](#), [tda_quant\(\)](#), [tda_rcorr\(\)](#), [tda_segr\(\)](#), [tda_subm\(\)](#)

Examples

```
set.seed(26)
d <- data.frame(x = rnorm(40), y = rnorm(40))
d$y <- d$y + 0.6 * d$x
tda_cov(d)$matrix
cov(d) # matches
```

tda_cutree	<i>Cut a dendrogram into groups</i>
------------	-------------------------------------

Description

Builds a dendrogram with TDA's `hcl`s and cuts it with `hclsp`.

Usage

```
tda_cutree(
  d,
  nlev = 2,
  nodes = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

<code>d</code>	a <code>dist</code> , or a square dissimilarity matrix.
<code>nlev</code>	number of levels at which to cut.
<code>nodes</code>	an explicit sequence of node numbers to cut at, TDA's <code>cn=</code> .
<code>options</code>	a named list of further TDA options, passed through.
<code>dir</code>	working directory.
<code>...</code>	passed to tda_run .

Details

Unlike R's `cutree`, which returns one group label per observation, `hclsp` reports the cut as a set of nodes: `table` has one row per resulting cluster, its TDA node number, how many leaves (original observations) it holds, and its parent node – not a per-observation vector. Getting a label for each observation means walking merge (the dendrogram `hcl`s built, one row per step) down from a node in `table` to the leaves under it; this wrapper does not do that walk for you.

Value

An object carrying `merge`, the dendrogram, and `table`, one row per cluster in the cut (node, leaves, parent).

See Also

Other clustering: [TDA_CLUSTER](#), [TDA_MDS](#), [tda_conjoint\(\)](#), [tda_dma\(\)](#), [tda_pdatd\(\)](#)

Examples

```
set.seed(35)
pts <- rbind(matrix(rnorm(10, 0), 5, 2), matrix(rnorm(10, 8), 5, 2))
ct <- tda_cutree(dist(pts), nlev = 2)
ct$stable    # two clusters of five leaves each
ct$merge     # the nine merge steps that built the dendrogram
```

tda_cwt_norm	<i>Rescale case weights the way TDA's cwt(wnorm=) does</i>
--------------	--

Description

tda_cwt_norm reproduces TDA's `cwt(wnorm=s)=W`; (or the bare-flag `cwt(wnorm)=W`;) as a plain vector transform, for weights= on [tda_qreg](#), [tda_glm](#), [tda_ple](#), [tda_ltb](#) and [tda_rate](#) – none of which expose `wnorm` as an argument, since this is all it does: fitted coefficients come back identical whether the weights are rescaled this way first or not, only the standard errors change (rescaling is a constant multiplier on the whole weighted log-likelihood, so it moves the curvature at the optimum, not where the optimum is). `cwt(wnorm=s)=W`; on the raw weights and plain `cwt=W`; on `tda_cwt_norm(W, s)` give the same fit, to the last printed digit.

Usage

```
tda_cwt_norm(w, s = length(w))
```

Arguments

<code>w</code>	a numeric vector of raw case weights.
<code>s</code>	the target sum for the rescaled weights – TDA's <code>wnorm=s</code> . Left at its default, <code>length(w)</code> (the number of cases), it matches the bare-flag form, <code>cwt(wnorm)=W</code> ;, TDA's "keep the same effective sample size" correction.

Value

`w`, rescaled so it sums to `s`.

Examples

```
w <- c(2, 2, 2, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10) # sums to 74
tda_cwt_norm(w)      # cwt(wnorm)=W;    -- rescaled to sum 13 (length(w))
tda_cwt_norm(w, 100) # cwt(wnorm=100)=W;
```

tda_dblock	<i>Declare a block structure</i>
------------	----------------------------------

Description

TDA's standalone dblock on the current-run data: blocks are maximal runs of identical values. Mostly useful inside longer [tda_run](#) pipelines; for block-mode variable creation put dblock= inside nvar (see the package tests).

Usage

```
tda_dblock(data, by, commands = character(), ...)
```

Arguments

data	data frame.
by	column name defining blocks.
commands	further command lines run under the block structure. Column names are TDA-sanitised (upper-cased) first, so refer to them in that form.
...	passed to tda_run .

Value

the printed output, invisibly.

Examples

```
out <- tda_dblock(data.frame(G = c(1, 1, 2), X = 1:3), by = "G",
                  commands = "dstat = X;")
# blocks are defined by G; dstat runs under the block structure
grep("^X", out, value = TRUE)
```

tda_derive	<i>Derive variables with TDA's expression language</i>
------------	--

Description

Runs nvar on a data frame with new variables defined by TDA expressions, and returns the frame with them added. This is the way to the operators of the manual's 5.2.6 that have no R counterpart: the change and count operators (change, cntch), the aggregates over blocks (gcnt, grec, gmean, gstd, ...), TDA's random numbers, and so on.

Usage

```
tda_derive(data, ..., block = NULL, options = list(), dir = tempfile("tda"))
```

Arguments

<code>data</code>	a data frame.
<code>...</code>	named TDA expressions, as strings: <code>CH = "change(S)"</code> .
<code>block</code>	a column name: with it, <code>nvar</code> runs in block mode (<code>dblock=</code>), and the block operators aggregate within blocks of consecutive rows with the same value.
<code>options</code>	a named list of further <code>nvar</code> options.
<code>dir</code>	working directory.

Value

data with the new columns, from the export channel.

Examples

```
d <- data.frame(ID = c(1, 1, 2), S = c(1, 3, 2))
tda_derive(d, CH = "change(S)", CN = "cntch(S)")
tda_derive(d, CN = "cntch(S)", block = "ID")
```

<code>tda_diagnostics</code>	<i>Diagnostics TDA printed during a run</i>
------------------------------	---

Description

Every error, warning, note or "cannot" line TDA printed, on stdout or stderr, for a run or a fitted model. TDA reports problems by printing them and carrying on, so a fit can come back with a full table and a message about what it could not do; nothing here is dropped from `tda_output()`, this is the short list.

Usage

```
tda_diagnostics(x)
```

Arguments

<code>x</code>	a <code>tda_result</code> from <code>tda_run()</code> or any fitted model.
----------------	--

Value

a character vector, empty when TDA reported nothing.

Examples

```
# a clean run reports nothing
tda_diagnostics(tda_run("nvar(noc = 5, X = case);"))

# dstat asks for a variable that was never defined: TDA prints the
# error and continues, so the run returns and the message is here
r <- tda_run(c("nvar(noc = 5, X = case);", "dstat = Y;"))
tda_diagnostics(r)

# a fit stopped after one iteration still returns a coefficient
# table; the problem is reported alongside it
d <- data.frame(x = c(-1, -0.5, 0, 0.5, 1, 1.5), y = c(0, 0, 1, 0, 1, 1))
f <- suppressWarnings(tda_qreg(y ~ x, d, control = tda_control(maxit = 1)))
tda_diagnostics(f)
```

tda_dma

Principal components, factor analysis, correspondence analysis

Description

dma: seven related methods for reducing a data matrix to a handful of derived scores, selected with `alg` – principal components (1, from the raw data; 2, from a covariance or correlation matrix already computed), factor analysis (3, 4, the same two starting points but with eigenvectors rescaled as loadings), dual scaling and correspondence analysis (5, 6, for a frequency table rather than a data matrix), and a direct SVD-based projection (7). Verified against [prcomp](#) for algorithm 1: eigenvalues match exactly, eigenvectors match up to sign (an ordinary, harmless ambiguity in any eigendecomposition).

Usage

```
tda_dma(
  data,
  alg = 1,
  preprocess = NULL,
  ns = NULL,
  variables = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

<code>data</code>	a data frame or matrix. For algorithms 1–4 and 7, a standard data matrix (rows are cases); for 2 and 4, a covariance or correlation matrix (square, symmetric); for 5 and 6, a frequency table.
<code>alg</code>	which of the seven algorithms; see Description. Default 1, ordinary principal components.

preprocess	preprocessing, for algorithms 1, 5 and 7 only – for alg = 1: 1 nothing (TDA's default), 2 mean centre, 3 standardize; for alg 5 or 7: 2/3 convert to row or table relative frequencies.
ns	keep at most this many components; all of them by default.
variables	optional subset of data's column names to use, instead of all of them.
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to <code>tda_run</code> . Further dma options can be given the same way, notably <code>opt=</code> (algorithm variant) and <code>pcf=</code> (print classification frequencies).

Details

TDA's `v=` (choosing a subset of variables from a larger data matrix) needs variable names that start uppercase, the same rule as anywhere else in TDA – lowercase R column names are displayed correctly when declared but cannot be *referenced* again later in the same script, including by `v=` itself, failing with a “Syntax error or undefined variables” that gives no hint the name's case is the problem. `variables` handles this automatically; there is no need to rename columns yourself first.

Value

A list with values (the eigenvalues), percent (each eigenvalue's share of the total, as TDA prints it), vectors (the eigenvectors or factor loadings, one column per component, row names taken from data), and scores (the derived score for each case, one column per component) – scores is empty for algorithms where TDA does not calculate it (2 and 4).

See Also

Other clustering: `TDA_CLUSTER`, `TDA_MDS`, `tda_conjoint()`, `tda_cutree()`, `tda_pdatd()`

Examples

```
set.seed(1)
d <- data.frame(x1 = rnorm(30), x2 = rnorm(30))
d$x3 <- d$x1 + d$x2 + rnorm(30, sd = 0.3) # correlated with both
pc <- tda_dma(d)
pc$values          # most of the variance is in the first PC
pc$vectors         # loadings: x3 dominates it
head(pc$scores)

# a subset of the variables, instead of all of them
tda_dma(d, variables = c("x1", "x2"))$values
```

tda_dmet	<i>Convert a distance matrix into a metric</i>
----------	--

Description

dmet: an undirected valued graph's edge values, read as a distance matrix, do not always satisfy the triangle inequality (a direct edge can be "longer" than going via a third node). This finds the smallest constant that, added to every distance, makes the whole matrix satisfy it – $c = \max(0, \max_{i,j,k} d_{ij} - d_{ik} - d_{kj})$ – and returns both the original and the corrected matrix.

Usage

```
tda_dmet(g, options = list(), dir = tempfile("tda"), ...)
```

Arguments

g	a tda_graph , undirected and valued.
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to tda_run .

Value

A list: original and modified, both distance matrices, and constant, the value added to get from one to the other.

See Also

Other graph analysis: [plot.tda_graph\(\)](#), [tda_g\(\)](#), [tda_g_analyses](#), [tda_graph\(\)](#), [tda_locate_line\(\)](#), [tda_ptree\(\)](#), [tda_ucl\(\)](#)

Examples

```
e <- data.frame(from = c(1, 2, 3), to = c(2, 3, 1), value = c(1, 2, 5))
g <- tda_graph(e, directed = FALSE) # 1-3 direct (5) exceeds
                                     # 1-2-3 via node 2 (1+2=3)
tda_dmet(g)
```

tda_dmet1	<i>Distances made metric</i>
-----------	------------------------------

Description

TDA’s dmet1: adjusts a distance matrix (given as an undirected valued graph) toward metricity.

Usage

```
tda_dmet1(d, tolerance = 1e-04, max_iterations = 20, ...)
```

Arguments

- d symmetric distance matrix.
- tolerance, max_iterations deviation tolerance and cap.
- ... passed to [tda_run](#).

Value

the output file’s lines.

Examples

```
# 5 breaks the triangle inequality (1 + 2 < 5), so the
# metricized matrix shortens it to 3
d <- rbind(c(0, 1, 5), c(1, 0, 2), c(5, 2, 0))
r <- tda_dmet1(d)
r$modified
```

tda_dple	<i>Life tables, Kaplan-Meier, and the discrete time estimators</i>
----------	--

Description

tda_ple is the product-limit estimator, which is Kaplan-Meier; tda_km is an alias for it. tda_ltb groups into intervals. tda_dple and tda_dltb are the discrete time versions, and tda_diple handles interval censored data.

Usage

```

tda_dple(formula, data, dir = tempfile("tda"), ...)

tda_dltb(formula, data, dir = tempfile("tda"), ...)

tda_diple(data, start, lower, upper, status, dir = tempfile("tda"), ...)

tda_ltb(
  formula,
  data,
  tp = NULL,
  cfrac = NULL,
  weights = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)

tda_ple(
  formula,
  data,
  compare = FALSE,
  quantiles = NULL,
  at_survival = NULL,
  weights = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)

tda_km(formula, data, ...)

```

Arguments

formula	Surv(t, s) ~ 1, or with a grouping term on the right to estimate separately per group.
data	a data frame.
dir	working directory.
...	passed to tda_run .
start, lower, upper, status	columns for tda_diple, as names or as vectors. The event is known to fall between lower and upper. Explicit columns rather than a formula, because the four-argument Surv() already means (start, end, origin, destination) for the multi-state models.
tp	interval boundaries for the life table, e.g. seq(0, 500, 30).
cfrac	for tda_ltb, the fraction of censored cases within an interval counted as part of the risk set (TDA's cfrac=, default 0.5).

weights	optional case weights, a column name or a vector as long as the data – TDA's <code>cwt = W;</code> , a separate command rather than an <code>edef=</code> option, which stays active for whatever runs after it.
options	a named list of further TDA options, passed through.
compare	for <code>tda_ple</code> , also run TDA's tests of whether the group survivor functions differ. Needs at least two groups.
quantiles	for <code>tda_ple</code> , time values to report the survivor function at, e.g. <code>c(5, 10, 20);</code> TDA's <code>qt=</code> . The resulting table (one row per transition per quantile) is parsed into the returned object's <code>quantiles</code> component, not left as raw console text – <code>qt=</code> writes to the console, not <code>out.ple</code> , unlike every other table <code>tda_ple</code> returns.
at_survival	for <code>tda_ple</code> , the reverse of <code>quantiles</code> : survival probabilities, in decreasing order (<code>seq(0.9, 0.1, by = -0.1)</code>), or just <code>0.5</code> for the median survival time), to report the time each is reached at – TDA's <code>qo=</code> . Despite an identical description in TDA's help text, this is a different request from <code>quantiles/qt=</code> , not an alias for it. Several values are fine, in decreasing order: TDA's parser reads <code>qo=</code> with the same list reader as <code>qt=</code> and rejects an ascending list as a syntax error, which is what an earlier note here mistook for "one value only".

Details

The survivor function's standard errors are the usual Greenwood estimate (the manual gives the formula in section 6.5.2), the same one `survival::survfit` reports, so the two agree on identical data.

Value

An object carrying `table`, `blocks` with one entry per group or transition, and for the product limit median. Be precise about the pair: `table` is deliberately only the *first* block, the convenience view for the common single-group call; a grouped or multi-transition fit's full result is `blocks` (named by group), and anything that must see everything – comparisons, exports, your summaries – should read `do.call(rbind, fit$blocks)`, never `table` alone. The product-limit columns are `time`, `events`, `censored`, `n.risk`, `survivor`, `std.err` and `cum.rate`.

Censoring and competing risks

The second argument of `Surv()` is the destination state, not simply an event indicator. Zero means no transition, so it is a censored case. A value above 1 is a second destination, which makes the data competing risks, and the estimate is returned per transition in `$blocks`.

See Also

Other rate models: [tda_constrain\(\)](#), [tda_control\(\)](#), [tda_rates\(\)](#), [tda_split\(\)](#), [tda_survivor\(\)](#), [tda_transitions\(\)](#), [vcov.tda_fit\(\)](#)

Examples

```
d <- data.frame(t = c(4, 3, 1, 1, 2, 2, 3, 5), s = c(1, 1, 1, 0, 1, 1, 0, 1))
km <- tda_km(Surv(t, s) ~ 1, d)
km$table
```

```

plot(km)

# ltb: the same data grouped into intervals rather than at each event time
tda_ltb(Surv(t, s) ~ 1, d, tp = seq(0, 6, 2))$table

# ple: tda_km's underlying function, with two features tda_km does
# not expose -- compare = TRUE tests whether group survivor functions
# differ, and quantiles reads the curve at chosen times
set.seed(1)
d1b <- data.frame(t = round(rexp(40, 0.1), 1) + 0.5, s = rbinom(40, 1, 0.8),
                  grp = rep(c("A", "B"), 20))
pf <- tda_ple(Surv(t, s) ~ factor(grp), d1b, compare = TRUE,
              quantiles = c(5, 10))
pf$comparison # a log-rank test between the two groups' curves
pf$quantiles  # the survivor function at t = 5 and t = 10, per group

# dple, dlbtb: the discrete-time versions, for integer time already grouped
# into periods -- t itself is the period here, not a continuous duration
set.seed(44)
d2 <- data.frame(t = sample(1:10, 60, TRUE), s = rbinom(60, 1, 0.8))
tda_dple(Surv(t, s) ~ 1, d2)$table
tda_dlbtb(Surv(t, s) ~ 1, d2)$table

# dipple: the event is known only to fall between lower and upper
set.seed(45)
d3 <- data.frame(start = 0, lower = sample(1:8, 40, TRUE))
d3$upper <- d3$lower + sample(1:3, 40, TRUE)
d3$status <- rbinom(40, 1, 0.85)
tda_dipple(d3, start = "start", lower = "lower", upper = "upper",
           status = "status")$table

```

tda_dstat

*Descriptive statistics***Description**

These take data rather than a formula, following `cor` and `summary`: there is no response and no model here. A data frame, a matrix, or loose vectors are all accepted, and non-numeric columns are dropped.

Usage

```
tda_dstat(..., by = NULL, options = list(), dir = tempfile("tda"))
```

```
tda_corr(..., options = list(), dir = tempfile("tda"))
```

Arguments

... a data frame, a matrix, or vectors.

- by optional grouping, as a vector or a one-sided formula, in the manner of aggregate. TDA prints one table per group and they are returned stacked with a Group column.
- options a named list of further TDA options, passed through.
- dir working directory.

Details

tda_dstat reports each variable’s minimum, maximum, mean, standard deviation (denominator $n - 1$, or the weight total minus one under case weights) and sum. tda_corr and tda_cov are the ordinary product-moment correlation and the $n - 1$ -denominator covariance – cor() and cov() – extended to case weights when a cwt command is active (manual, section 6.2.6). All of them compute over every case given, including any -5 standing in for an NA – see ?tdaR on missing values.

Value

An object carrying a table, or for tda_corr a matrix.

See Also

Other descriptive statistics: [tda_brr\(\)](#), [tda_cov\(\)](#), [tda_freq\(\)](#), [tda_independence\(\)](#), [tda_ineq\(\)](#), [tda_loglin\(\)](#), [tda_quant\(\)](#), [tda_rcorr\(\)](#), [tda_segr\(\)](#), [tda_subm\(\)](#)

Examples

```
set.seed(1)
d <- data.frame(x = round(rnorm(40, 10, 3), 1))
d$y <- round(2 + 0.8 * d$x + rnorm(40, sd = 2), 1)
tda_dstat(d)
tda_corr(d$x, d$y)
```

tda_eigen	<i>Eigenvalues of a matrix, via TDA</i>
-----------	---

Description

Runs TDA’s etest, which computes the eigenvalues and eigenvectors of a square matrix and reports the residual of each. It is a test of TDA’s eigen routines rather than a general facility – [eigen](#) is what you want for ordinary work – but it is the way to check what TDA itself computes for a matrix a model is about to be fitted on.

Usage

```
tda_eigen(
  x,
  algorithm = 1,
  max_iter = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

<code>x</code>	a square numeric matrix.
<code>algorithm</code>	1 (default), 2 (eigen1) or 3 (eigen2).
<code>max_iter</code>	for <code>algorithm = 3</code> : iteration limit, TDA's default is 100.
<code>options</code>	a named list of further TDA options, passed through.
<code>dir</code>	working directory.
<code>...</code>	passed to <code>tda_run</code> .

Details

The eigenvalues come from an exporter rather than from the printed table, so they arrive at full precision instead of rounded to the print format.

Value

A data frame with one row per eigenvalue: `re` and `im`, its real and imaginary parts. `attr(x, "run")` carries the run, whose output holds the eigenvectors and residuals.

Examples

```
m <- diag(c(2, 3, 5))
tda_eigen(m)
```

<code>tda_episodes</code>	<i>The episode data behind a model</i>
---------------------------	--

Description

`tda_episodes` returns the episode data `edef` builds from a formula, which is what every model in this package is actually fitted to. It is the way to check that a `Surv()` specification means what you intended before trusting a fit.

Usage

```

tda_episodes(
  formula,
  data,
  vars = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)

tda_state_dist(
  formula,
  data,
  times,
  id = NULL,
  spell = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)

```

Arguments

formula	the same <code>Surv()</code> formula the model functions take.
data	a data frame.
vars	additional variables to carry into the output, as names.
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to tda_run .
times	time points at which to evaluate the state distribution.
id, spell	for <code>tda_state_dist</code> , columns identifying the case and numbering its spells, for multi-episode data (edef's <code>id=</code> and <code>sn=</code>).

Details

`tda_state_dist` gives the distribution over states at a set of time points – how many cases occupy each state when – which is the natural summary of multi-state data.

Value

A data frame of episodes, or of state proportions by time.

See Also

Other episodes: [tda_esort\(\)](#)

Examples

```
d <- data.frame(t = c(4, 3, 1, 5), s = c(1, 0, 1, 1), x = c(2, 1, 3, 1))
tda_episodes(Surv(t, s) ~ x, d)
tda_state_dist(Surv(t, s) ~ x, d, times = c(1, 3, 5))
```

tda_esort

*TDA's file utilities***Description**

TDA was written when a data file could be larger than the machine's memory, and these commands work a record at a time on an ASCII file rather than loading it: esort sorts the records, eskip drops character positions, eselect keeps the records whose key appears in a second file, emerge joins sorted files on a key. They are file-in, file-out, on raw text records addressed by character position, and they are that here too: name the input and the output file and the output path comes back, invisibly, with the run as its "run" attribute. Nothing is read back; for a data frame, use [tda_read_table](#) on the file you asked for.

Usage

```
tda_esort(file, keys, out, options = list(), dir = tempfile("tda"))
```

```
tda_eskip(file, drop, out, options = list(), dir = tempfile("tda"))
```

```
tda_eselect(
  file,
  with,
  keys,
  with_keys = keys,
  out,
  options = list(),
  dir = tempfile("tda")
)
```

```
tda_emerge(
  file,
  with,
  keys,
  with_keys = keys,
  out,
  options = list(),
  dir = tempfile("tda")
)
```

```
tda_ejoin(
  data,
  id,
```

```

    start,
    end,
    state,
    with = NULL,
    with_id = NULL,
    with_start = NULL,
    with_end = NULL,
    with_state = NULL,
    options = list(),
    dir = tempfile("tda")
  )

```

Arguments

file	an existing ASCII data file.
keys	character positions in the record, as a start and end pair – <code>c(1, 3)</code> is the first three characters. Required: they are positions in the line rather than column numbers. Several keys are given as a list of pairs, up to five.
out	the output file to write.
options	a named list of further TDA options, passed through.
dir	working directory.
drop	character positions to drop, in the same form.
with	a second file supplying the keys, or the file(s) to merge in.
with_keys	where the key sits in with, defaulting to the same positions as in file. <code>eselect</code> and <code>emerge</code> need a range in each file.
data	for <code>tda_ejoin</code> , a data frame.
id, start, end, state	for <code>tda_ejoin</code> , the episode columns in data: case identifier, spell start and end, and the state during the spell.
with_id, with_start, with_end, with_state	for <code>tda_ejoin</code> , the same four columns in with, when with is given and its own column names differ from data's; default to id/ start/end/state's values.

Details

`tda_emerge` is a merge join and requires both inputs sorted in ascending order on the key; it says so and stops otherwise, naming the two records that are out of order.

`tda_ejoin` is different in kind: it joins episode data into levels, takes data frames, and returns one through the export channel.

Value

The output path, invisibly, for the four file utilities; a data frame for `tda_ejoin`.

See Also

Other episodes: [tda_episodes\(\)](#)

Examples

```
f <- tempfile(); writeLines(c("103 30", "101 10", "102 20"), f)
o <- tempfile()
tda_esort(f, keys = c(1, 3), out = o)
readLines(o)
tda_eskip(f, drop = c(1, 4), out = o)
readLines(o)

# eselect/emerge need their inputs already sorted on the key
f1 <- tempfile(); writeLines(c("101 10", "102 20", "103 30"), f1)
f2 <- tempfile(); writeLines(c("101 1", "103 3"), f2)
tda_eselect(f1, with = f2, keys = c(1, 3), out = o); readLines(o)
tda_emerge(f1, with = f2, keys = c(1, 3), out = o); readLines(o)

# ejoin: joins one or two episode datasets, splitting overlapping
# spells into levels -- an activity spanning 0-10, and a state that
# changes mid-activity at t=4, joined so each row is a period where
# neither changes
act <- data.frame(id = 1, start = 0, end = 10, state = 1)
st <- data.frame(id = 1, start = c(0, 4), end = c(4, 10), state = c(1, 2))
tda_ejoin(act, id = "id", start = "start", end = "end", state = "state",
          with = st)
```

tda_evalfi

Evaluate an inclusion function over an interval

Description

evalfi: given an expression and an interval domain for each of its arguments, returns an interval guaranteed to contain the expression's true range over that domain – interval arithmetic, not a numeric evaluation at a point. The bound is conservative rather than tight where a variable appears more than once (the standard "dependency problem" of interval arithmetic: each occurrence is widened independently, as if they were unrelated variables that happened to share a domain) – evalfi($x = c(0, 3)$, " $x*x-2*x+1$ ") returns $[-5, 10]$, not the true range $[0, 4]$ that $(x-1)^2$ takes over $[0, 3]$. evalfi1 is the same but for the first derivative.

Usage

```
tda_evalfi(
  expr,
  ...,
  derivative = FALSE,
  options = list(),
  dir = tempfile("tda")
)
```

Arguments

<code>expr</code>	the expression, as a string, in TDA's language.
<code>...</code>	one interval per argument the expression uses, named for the argument, each as <code>c(lower, upper) – evalfi(x = c(0, 3), expr = "...")</code> .
<code>derivative</code>	if TRUE, use <code>evalfi1</code> (the first derivative's inclusion function) instead of <code>evalfi</code> .
<code>options</code>	a named list of further TDA options, passed through.
<code>dir</code>	working directory.

Details

$\wedge$ is not a valid operator in an interval expression – write a power as repeated multiplication ($x*x$, not x^2).

Value

A named numeric vector, `c(lower, upper)`.

See Also

Other sequence analysis: `tda_fml()`, `tda_frml()`, `tda_seq_info()`, `tda_seqgc()`, `tda_seqm()`, `tda_seqmd()`, `tda_seqpe()`, `tda_seqpm()`

Examples

```
# true range of (x-1)^2 over [0,3] is [0,4] -- this is wider, since x
# appears twice and interval arithmetic cannot see they are the same x
tda_evalfi("x*x-2*x+1", x = c(0, 3))
tda_evalfi("x*x-2*x+1", x = c(0, 3), derivative = TRUE) # 2x-2, exact
                                                         # here: [-2,4]
```

 tda_expressions | *Expand an expression, or parse a matrix expression* |**Description**

`tda_expand` runs TDA's `expm`, which expands an expression into its fully written-out form – a model formula with interaction shorthand becomes the explicit list of terms. `tda_mparse` runs `mparse`, which parses a matrix expression and reports how it read it, without evaluating anything.

Usage

```
tda_expand(expr, setup = NULL, options = list(), dir = tempfile("tda"), ...)
```

```
tda_mparse(
  expr,
  setup = NULL,
```

```

    mfmt = NULL,
    options = list(),
    dir = tempfile("tda"),
    ...
)

```

Arguments

<code>expr</code>	the expression, as a single string.
<code>setup</code>	TDA commands to run first, as a character vector. A matrix expression needs its matrices to exist, so this is where <code>mdef</code> / <code>mdefi</code> lines go.
<code>options</code>	a named list of further TDA options, passed through.
<code>dir</code>	working directory.
<code>...</code>	passed to <code>tda_run</code> .
<code>mfmt</code>	for <code>tda_mparse</code> : print format, TDA's default is 12.4.

Details

Both are diagnostics: they answer "what did TDA make of what I wrote", which is otherwise only visible in an error message.

Value

A character vector: the lines TDA printed in response. `attr(x, "run")` carries the run.

Examples

```

tda_expand("A*B")
tda_mparse("A+B", setup = c("mdefi(2,2,A);", "mdefi(2,2,B);"))

```

TDA_FAMILIES

Generalized linear models

Description

Fits a generalized linear model, taking R's family objects.

Usage

```

TDA_FAMILIES

tda_glm(
  formula,
  data,
  family = stats::gaussian,
  link = NULL,
  custom_link = NULL,

```

```

domain = NULL,
trials = NULL,
weights = NULL,
intercept = NULL,
start = NULL,
predictions = FALSE,
protocol = FALSE,
equality = NULL,
inequality = NULL,
control = NULL,
options = list(),
dir = tempfile("tda"),
...
)

```

Arguments

formula	a two-sided formula.
data	a data frame.
family	an R family object, a family function, its name, or TDA's own d= number (1-5) directly – see Families and links below for exactly which ones, and which link functions, TDA itself implements.
link	the link function – a name (from Families and links below) or TDA's link= number (1-8) directly, matching <code>tda_qreg</code> 's model/kernel convention. Overrides whatever link family itself specifies (an R family object's link=), when both are given; NULL (the default) defers to family's link, or the first one listed for it if family does not say. Lets a link be picked without constructing a full R family object for it, e.g. family = "Gamma", link = 2 instead of family = Gamma(link = "log") – both reach the identical TDA call. Ignored if custom_link is given.
custom_link	a user-defined link function, one not among the 8 in link – glm's right-hand side () = expression;) instead of a numbered link=, a TDA expression written in terms of a predefined variable named mue (the fitted mean; not something to declare, TDA recognises it automatically in this one context).
domain	the domain custom_link is valid on – TDA's ab=, default c(0, 1) if not given. Only meaningful together with custom_link.
trials	for a binomial model, the number of trials, with the response given as the number of successes (not a proportion) – glm's yw=. Not a general case weight for other families; TDA ignores it if given for one.
weights	optional case weights, a column name in data or a vector as long as the data – TDA's cwt = W; , the same mechanism and convention as <code>tda_ple</code> 's weights. Can be used together with trials, for a binomial-trials model where some cases should also count for more than others: trials shapes the binomial likelihood itself, weights separately scales each case's contribution to the total on top of that. TDA's cwt(wnorm=s)=W rescales the weights before use without changing the fitted coefficients, only the standard errors – not exposed as a separate argument here, since it is exactly reproduced by rescaling weights itself before passing it in (weights * s / sum(weights)).

intercept	NULL (the default) defers to formula itself, $y \sim x - 1$ already meaning no intercept the ordinary R way – glm's <code>ni=</code> .
start	optional starting values, in the order the predictors are listed – glm's <code>xp=</code> .
predictions	also write out each case's fitted mean, linear predictor, and predictor values – glm's <code>pres=/ dtda=</code> . Different in shape from both <code>tda_qreg</code> 's own predictions (categorical outcome probabilities) and <code>tda_fml</code> 's residuals (one value per case): one row per case, with <code>Mue</code> the fitted mean on the response scale (<code>R's fitted(fit, type = "response")</code>), <code>Eta</code> the linear predictor (<code>predict(fit, type = "link")</code>), each predictor's value, and the working weight. In <code>fit\$predictions</code> .
protocol	ask TDA to also write its iteration-by-iteration diagnostic log – glm's <code>prot=</code> , the same mechanism as <code>tda_fml</code> 's protocol. In <code>fit\$protocol</code> .
equality, inequality	linear constraints on the coefficients, written in the predictors' own names – the same mechanism and syntax as <code>tda_lsreg</code> 's equality/ inequality (<code>lsecon=/lsicon=</code>).
control	convergence settings from <code>tda_control</code> .
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to <code>tda_run</code> .

Value

An object of class `tda_fit`, its `fit` element (shown by `summary()`) carrying the deviance and residual degrees of freedom, the rank of the design matrix, and – for the families where TDA prints them – the Pearson statistic and the ML- and deviance-based dispersion estimates. Cases with missing values are dropped before the fit, with a message (see `?tdaR`). `logLik()` is NA: TDA's `glm()` never prints a log-likelihood itself, only Deviance – and it stays that way here rather than being reconstructed from Deviance, since that reconstruction would be a derived, unverifiable-by-the-user computation on top of what TDA itself actually computes.

Families and links

TDA_FAMILIES maps R's names onto TDA's `d=` (distribution) and `link=` numbering, from TDA's manual (which `tda_help("glm")` cannot show – `glm` has no entry in TDA's help database, despite being fully implemented). The first link listed for each family is the default, and is the canonical one:

family	d=	links (name = TDA's link=)
gaussian	1	identity=1, log=2, inverse=4
binomial	2	logit=3, probit=5, cloglog=6
poisson	3	log=2, identity=1, sqrt=7
Gamma	4	inverse=4, identity=1, log=2
inverse.gaussian	5	1/mu^2=8, inverse=4, identity=1, log=2

TDA's link numbering runs 1 to 8 in total (identity, log, logit, reciprocal/inverse, probit, cloglog, sqrt, and the inverse-Gaussian's quadratic inverse, $1/\mu^2$) – every one of the 8 appears above, on whichever family it is offered for through R's `family()` system. A family TDA does not implement,

or a link it does not offer for that family, is rejected with the list of what is available rather than silently replaced by a default.

glm's `ab=` option – the domain for a user-defined link function, one not among the 8 above, given as its TDA expression on the right-hand side of the command rather than a number – is reachable through `custom_link/domain` below.

See Also

Other regression: [TDA_QRMODELS](#), [tda_freg\(\)](#), [tda_gdf\(\)](#), [tda_l1reg\(\)](#), [tda_lsreg\(\)](#), [tda_mlrc_design\(\)](#), [tda_mreg\(\)](#), [tda_nlreg\(\)](#), [tda_npreg\(\)](#), [tda_zreg\(\)](#)

Examples

```
set.seed(5)
d <- data.frame(x = round(rnorm(40), 2))
d$y <- rbinom(40, 1, plogis(-0.5 + 1.2 * d$x))
fit <- tda_glm(y ~ x, d, family = binomial)
fit
predict(fit)                                # fitted values, on the data
                                           # the model was fit on
predict(fit, newdata = data.frame(x = 1))

# a binomial model of trial data (successes out of a known number of
# trials), not just 0/1 outcomes -- the response is the success count,
# trials the number of trials, not a proportion
set.seed(1)
n <- 60
d2 <- data.frame(x = rnorm(n))
p <- plogis(0.3 + 0.8 * d2$x)
d2$trials <- sample(5:20, n, replace = TRUE)
d2$successes <- rbinom(n, d2$trials, p)
coef(tda_glm(successes ~ x, d2, family = binomial, trials = d2$trials))

# case weights instead (or as well) -- a different thing from
# the trial counts above
d2$imp <- runif(n, 0.5, 2)
coef(tda_glm(successes ~ x, d2, family = binomial, trials = d2$trials,
              weights = "imp"))

# a constrained fit: the two coefficients forced to sum to exactly 1
d3 <- data.frame(x1 = rnorm(60), x2 = rnorm(60))
d3$y <- rbinom(60, 1, plogis(0.6 * d3$x1 + 0.4 * d3$x2))
coef(tda_glm(y ~ x1 + x2, d3, family = binomial,
              equality = "x1 + x2 = 1"))

# family and link as TDA's raw numbers -- useful when translating
# a .cf file directly, which only ever writes them this way. Reaches
# the identical call as family = Gamma(link = "log").
set.seed(2)
d4 <- data.frame(x = rnorm(60))
d4$y <- rgamma(60, shape = 2, rate = 2 / exp(0.5 + 0.3 * d4$x))
coef(tda_glm(y ~ x, d4, family = 4, link = 2))
```

```
# a user-defined link -- the logit link, spelled out by hand instead
# of using link = "logit"; reaches the same fit exactly
coef(tda_glm(y ~ x, d, family = binomial,
            custom_link = "log(mue / (1 - mue))", start = c(0, 0)))
```

tda_fml

Maximum likelihood with a user-defined likelihood

Description

fml maximises a log-likelihood you write yourself, which is what TDA offers that a fixed catalogue of models cannot. The likelihood is written in an expression language, evaluated inside TDA rather than as an R function – but as of this version, that expression can be written in ordinary R syntax and translated automatically, rather than TDA's.

Usage

```
tda_fml(
  definitions,
  data,
  start = NULL,
  control = NULL,
  constraints = NULL,
  residuals = FALSE,
  residual_vars = NULL,
  protocol = FALSE,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

definitions	either an unevaluated { } block of R assignments (see the section on writing the likelihood) or a character vector of TDA's assignments, the last defining fn, the contribution of one case to the log-likelihood. Names that appear on the right but are neither variables in data nor defined earlier are the parameters to be estimated.
data	a data frame; its columns are the variables the definitions can refer to.
start	optional named vector of starting values, e.g. c(b0 = 0, b1 = 1).
control	convergence settings from tda_control .
constraints	optional linear constraints on the parameters – fml's con=, the same machinery tda_qreg 's constraints uses. Refers to parameters by their position, b1/b2/..., 1-based, in the order they are first introduced across definitions – never by whatever name was actually given them. A parameter named b0 in

	definitions is still b1 here if it is the first one introduced; TDA's error for a literal "b0 = ..." constraint is "Error in parameter index", since b0 is not a valid 1-based position. Run once without constraints and check <code>tda_estimates()</code> 's <code>Idx</code> column for which position is which parameter before writing one. Must be strictly fewer constraints than parameters (TDA's rule; its error reads "number of constraints should be less than number of parameters").
<code>residuals</code>	ask TDA to also compute, per case, its contribution to the log-likelihood at the converged parameters – <code>fml</code> 's <code>pres=</code> . Not a classical observed-minus-fitted residual – <code>fml</code> has no built-in notion of a fitted value to subtract from, unlike <code>tda_lsreg</code> 's residuals. For $fn = -0.5 * (y - xb)^2$, say, this is that expression's value per case, not $y - xb$ itself.
<code>residual_vars</code>	with <code>residuals = TRUE</code> , extra columns from data to write alongside <code>fn</code> 's value, one column per case, in the order given – <code>fml</code> 's <code>v=</code> . <code>\$residuals</code> then comes back as a data frame (<code>fn</code> plus each named column) instead of a plain vector. Has no effect without <code>residuals = TRUE</code> .
<code>protocol</code>	ask TDA to also write its iteration-by-iteration diagnostic log – <code>fml</code> 's <code>prot=</code> : the iteration table (function value, gradient norm, parameter change per step), the parameter vector, and the covariance matrix at convergence. Not one consistent shape across every algorithm and situation, so returned as <code>\$protocol</code> , the file's text lines, rather than force-parsed into a table that would not fit every case.
<code>options</code>	a named list of further TDA options, passed through.
<code>dir</code>	working directory.
<code>...</code>	passed to <code>tda_run</code> .

Value

An object of class `tda_fit`. With `residuals = TRUE`, also carries `$residuals` (see residuals above).

Writing the likelihood in R syntax

Give definitions as an unevaluated `{ }` block of plain R assignments instead of a character vector, and it is translated to TDA's own expression syntax for you – never actually run as R code (DES, rate and the rest only ever exist inside TDA, so it cannot be). `ifelse(cond, yes, no)` becomes TDA's `if()` (TDA's `if` takes a 2- or 3-argument `if(cond, then)/if(cond, then, else)` shape matching `ifelse` exactly, not R's control-flow `if`), and R's comparison operators (`<`, `>`, `<=`, `>=`, `==`, `!=`) become TDA's comparison functions (`lt`, `gt`, `le`, `ge`, `eq`, `ne`). Arithmetic (`+`, `-`, `*`, `/`, `^`), `&|`, and ordinary function calls (`exp`, `log`, `negbin`, ...) already mean the same thing in both and pass through unchanged.

These six TDA functions are not the strict comparisons their R equivalents are. Every one of them – `lt`, `le`, `gt`, `ge`, `eq`, `ne` alike – carries a fixed tolerance, `EPSI2 = 1000 * DBL_EPSILON` (about $2.22e-13$): `le`, say, computes $x \leq y + \text{EPSI2}$, not $x \leq y$. This can matter on real data: a case in `examples/exam/rrdat.1` has `PRESN/PRES - 1` equal to `0.2` exactly in exact arithmetic, and the two land on different sides of `0.2` once represented as IEEE 754 doubles – `ge(x, 0.2)` inside TDA still calls it true, R's own `x >= 0.2` on the identical computation does not. Applied consistently across all six, at least, but nowhere in TDA's documentation: not in the manual, not in `tda_help()`'s operator

table, which describes `le` as plainly as “true if less than, or equal” with no footnote at all. A hand-written R replication of a TDA calculation involving a comparison near a round number (`=`, or a boundary like `0.2` here) can therefore disagree with TDA’s output at that boundary, for a reason with nothing to do with which language computed it – this is a property of TDA’s expression evaluator, present whether the comparison was written by hand in TDA’s syntax or reached it through this translation.

```
tda_fml({
  rate = exp(a0 + COH02 * a1 + COH03 * a2 + W * a3)
  l1 = ifelse(DES, log(rate), 0)
  fn = l1 - rate * DUR
}, data = d, start = c(-4, 0, 0, 0))
```

The older, direct form – a character vector of TDA’s assignment syntax – still works exactly as before, for anything the translator does not (yet) cover, or for pasting a `.cf` file’s text in directly.

```
tda_fml(c("xb = b0 + x * b1",
  "fn = -0.5 * (y - xb)^2"), data = d)
```

estimates `b0` and `b1` by least squares written as a likelihood. TDA’s functions are available, so `fn = negbin(alpha, gamma, y)` and the rest work.

See Also

Other sequence analysis: [tda_evalfi\(\)](#), [tda_frml\(\)](#), [tda_seq_info\(\)](#), [tda_seqgc\(\)](#), [tda_seqm\(\)](#), [tda_seqmd\(\)](#), [tda_seqpe\(\)](#), [tda_seqpm\(\)](#)

Examples

```
set.seed(1)
d <- data.frame(x = 1:20, y = 2 + 0.5 * (1:20) + rnorm(20))
tda_fml(c("xb = b0 + x * b1", "fn = -0.5 * (y - xb)^2"), d)
```

```
# the same model, written in R syntax instead
tda_fml({
  xb = b0 + x * b1
  fn = -0.5 * (y - xb)^2
}, d)
```

```
# a compound Poisson (negative binomial) count regression -- TDA has no
# separate command for this (unlike its plain Poisson, which is a glm()
# family), but negbin(alpha, gamma, k) is a built-in log-likelihood
# function in its expression language, reachable through fml like any
# other: gamma is the mean, alpha = 1/sigma the dispersion
set.seed(7)
d2 <- data.frame(x = rnorm(200))
d2$ndi <- rnbinom(200, mu = exp(0.5 + 0.8 * d2$x), size = 2)
coef(tda_fml({
  xb = b0 + x * b1
  gamma = exp(xb)
```

```

    sigma = exp(sig)
    alpha = 1 / sigma
    fn = negbin(alpha, gamma, ndi)
  }, d2, start = list(b0 = 0, b1 = 0, sig = 0)))

```

tda_freg

Non-linear regression with a user-defined function

Description

freg fits a regression function you write yourself, in the same way [tda_fml](#) takes a likelihood. The expression is TDA's language, not R's.

Usage

```

tda_freg(
  definitions,
  data,
  start = NULL,
  control = NULL,
  constraints = NULL,
  residuals = FALSE,
  residual_vars = NULL,
  protocol = FALSE,
  options = list(),
  dir = tempfile("tda"),
  ...
)

```

Arguments

definitions	a character vector of TDA assignments, the last defining fn, the quantity to minimise – or an unevaluated { } block of plain R assignments instead, translated automatically the same way tda_fml 's definitions is (see its Details for exactly what gets translated).
data	a data frame; its columns are the variables the definitions may refer to.
start	optional named vector of starting values.
control	convergence settings from tda_control .
constraints	optional linear constraints on the parameters – freg's con=, the identical mechanism and bN-by-position convention as tda_fml 's constraints (see there for the full explanation and a worked example of the naming trap); it works the same way here.
residuals	ask TDA to also compute, per case, its contribution to fn at the converged parameters – freg's own pres=, the identical mechanism as tda_fml 's own residuals (see there for what it actually is: not a classical observed-minus-fitted residual). For fn = r*r, say, this is that squared residual's value, not r itself.

<code>residual_vars</code>	with <code>residuals = TRUE</code> , extra columns to write alongside <code>fn</code> 's value, one per case – <code>freg</code> 's own <code>v=</code> , the same mechanism as <code>tda_fml</code> 's <code>residual_vars</code> .
<code>protocol</code>	ask TDA to also write its iteration-by-iteration diagnostic log – <code>freg</code> 's <code>prot=</code> , the same mechanism as <code>tda_fml</code> 's <code>protocol</code> .
<code>options</code>	a named list of further TDA options, passed through.
<code>dir</code>	working directory.
<code>...</code>	passed to <code>tda_run</code> .

Details

`tda_nlreg`'s `expr` argument also fits a user-written nonlinear function now, using `nlreg`'s dedicated algorithm (proper standard errors, and orthogonal distance regression as an option) rather than a general-purpose minimiser – prefer it for an ordinary nonlinear least-squares fit. `tda_freg` stays the more flexible tool for the cases `nlreg` cannot express at all: any summed objective, not just a sum of squared residuals (a robust loss, a weighted or constrained one, anything written by hand).

Value

An object of class `tda_fit`. With `residuals = TRUE`, also carries `$residuals`.

Writing the function

`fn` is the quantity summed over cases and minimised, not the regression function itself. Least squares is therefore the squared residual:

```
tda_freg(c("r = y - a * exp(b * x)", "fn = r*r"), d,
        start = c(a = 1, b = 0.1))
```

Algorithm

TDA's default minimiser is Newton, which overflows `exp()` on the first step for functions of this shape. `tda_freg` therefore defaults to BFGS, which is stable and reaches the same estimates as `nls`. Pass `control = tda_control(algorithm = 5)` for TDA's default.

See Also

Other regression: `TDA_FAMILIES`, `TDA_QRMODELS`, `tda_gdf()`, `tda_l1reg()`, `tda_lsreg()`, `tda_mlr_design()`, `tda_mreg()`, `tda_nlreg()`, `tda_npreg()`, `tda_zreg()`

Examples

```
set.seed(1)
d <- data.frame(x = 1:20, y = 3 * exp(0.1 * (1:20)) + rnorm(20, 0, 0.5))
tda_freg(c("r = y - a * exp(b * x)", "fn = r*r"), d,
        start = c(a = 1, b = 0.1))

# the same model, written in R syntax instead
tda_freg({
  r = y - a * exp(b * x)
```

```
fn = r * r
}, d, start = c(a = 1, b = 0.1))
```

tda_freq

One- and two-way frequency tables

Description

tda_freq counts occurrences of each combination of values across one or more variables; tda_freq1 counts occurrences of each value of one variable; tda_freq2 cross-tabulates two. TDA's manual documents all three (freq, freq1, freq2) on one shared page, since they mostly share the same options (maxcat=/fmt=/tfmt=/df=) and differ mainly in shape. Their R outputs are shaped differently too – tda_freq/tda_freq1 return a table, tda_freq2 a matrix of counts – but all three describe a frequency distribution, so they sit together here as TDA's own manual does.

Usage

```
tda_freq(..., maxcat = NULL, dir = tempfile("tda"))

tda_freq1(..., maxcat = NULL, options = list(), dir = tempfile("tda"))

tda_freq2(
  ...,
  contingency = FALSE,
  maxcat = NULL,
  options = list(),
  dir = tempfile("tda")
)
```

Arguments

...	a data frame, a matrix, or vectors. tda_freq2 needs exactly two.
maxcat	the maximum number of categories per variable (all three commands' own maxcat=, default 1000).
dir	working directory.
options	a named list of further TDA options, passed through.
contingency	for tda_freq2, also calculate chi-square and the other contingency measures (freq2's sc=1, a flag rather than a real option – there is no other value). Reported in measures, a named numeric vector, not just printed.

Value

An object carrying a table for tda_freq/ tda_freq1, a matrix for tda_freq2, and, with contingency = TRUE, a measures named vector (chi-square, Cramer's V, lambda, Somers' D, and the rest).

See Also

Other descriptive statistics: [tda_brr\(\)](#), [tda_cov\(\)](#), [tda_dstat\(\)](#), [tda_independence\(\)](#), [tda_ineq\(\)](#), [tda_loglin\(\)](#), [tda_quant\(\)](#), [tda_rcorr\(\)](#), [tda_segr\(\)](#), [tda_subm\(\)](#)

Examples

```
set.seed(48)
d <- data.frame(g = sample(1:3, 80, TRUE), h = sample(1:2, 80, TRUE))
tda_freq1(d["g"])$table
tda_freq(d)$table # a joint table over both variables at once, unlike
                  # tda_freq1's single-variable count
tda_freq2(d)$matrix # a cross-tabulation, rows by g, columns by h
tda_freq2(d, contingency = TRUE)$measures[c("Cramer's V", "Phi")]
```

tda_frml

*Maximum likelihood for a user-defined transition rate model***Description**

frml: like [tda_fm1](#), but for episode (survival) data rather than plain cross-sectional rows – fn is the log-likelihood contribution of one *episode*, and can refer to time (the position within it) the way [tda_rate](#)'s own define can. This is not fm1 under another name: the same definitions that work with fm1 on plain rows fail here with “no episode data defined” until an actual episode structure (survival::Surv()) is given.

Usage

```
tda_frml(
  formula,
  data,
  definitions,
  start = NULL,
  id = NULL,
  spell = NULL,
  control = NULL,
  constraints = NULL,
  residuals = FALSE,
  residual_vars = NULL,
  protocol = FALSE,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

formula	a <code>Surv(...)</code> formula, the same as <code>tda_rate</code> 's.
data	a data frame.
definitions	a character vector of assignments, evaluated in order inside TDA, the last of which must define <code>fn</code> – see <code>tda_frml</code> for the same convention on plain data, and the description above for the episode-specific variables available here. Or an unevaluated <code>{ }</code> block of plain R assignments instead, translated automatically the same way <code>tda_frml</code> 's definitions is. Names on the right that are neither a covariate from formula nor one of TDA's built-ins, and are not defined earlier, are parameters to be estimated. formula's <code>Surv(...)</code> arguments can also be referenced directly by their real column name when they are a simple column reference (<code>Surv(TFP, DES)</code> 's <code>TFP/DES</code> , say, work the same as TDA's built-ins <code>tf - ts/des</code>) – matching how TDA's manual writes this (<code>examples/exam/frml1.cf</code> , referencing its own <code>DES/DUR</code> columns directly, not through any built-in alias). A computed <code>Surv()</code> argument (<code>Surv(TFin - TStart + 1, DES)</code>) has no one column to translate to, so only TDA's built-ins reach it.
start	optional named list of starting values, in the order the definitions introduce the parameters.
id, spell	columns identifying the case and numbering its spells, for multi-episode data – the same pair <code>tda_rate</code> takes; a case's spells then enter the likelihood together. <code>strata()</code> in the formula is refused rather than accepted: <code>frml</code> has no stratification mechanism and would silently ignore the term (fit per stratum instead).
control	optional <code>tda_control</code> .
constraints	optional linear constraints on the parameters – <code>frml</code> 's <code>con=</code> , the identical mechanism and <code>bN</code> -by-position convention as <code>tda_frml</code> 's constraints (see there for the full explanation and a worked example of the naming trap); it works the same way here.
residuals	ask TDA to also compute, per episode, its contribution to <code>fn</code> at the converged parameters – <code>frml</code> 's own <code>pres=</code> , the identical mechanism as <code>tda_frml</code> 's own residuals (see there for what it actually is: not a classical observed-minus-fitted residual), computed per episode here.
residual_vars	with <code>residuals = TRUE</code> , extra columns to write alongside <code>fn</code> 's value, one per episode – <code>frml</code> 's own <code>v=</code> , the same mechanism as <code>tda_frml</code> 's <code>residual_vars</code> . Only reliably supports a covariate from formula; TDA's episode built-ins (<code>ts</code> , <code>tf</code> , <code>org</code> , <code>des</code> , <code>time</code>) cannot be named here, despite being available inside <code>fn</code> : they are expression-only aliases, not real data-matrix variables – <code>v = ts</code> , <code>tf</code> is a TDA syntax error (their real, differently-named underlying variables, e.g. <code>Tstart</code> for <code>ts</code> , do work, but are internal, auto-generated by <code>edef()</code> and not something to build an interface around safely).
protocol	ask TDA to also write its iteration-by-iteration diagnostic log – <code>frml</code> 's <code>prot=</code> , the same mechanism as <code>tda_frml</code> 's <code>protocol</code> .
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to <code>tda_run</code> .

Details

fn's expression cannot reference the raw duration/status columns from data directly – `Surv()`'s columns are consumed by `edef()` into TDA's internal episode representation, and are not separately visible by name afterward. An fn written against the original column names fails with those names listed as *parameters to estimate* (TDA's convention for any identifier it does not otherwise recognise), not as a lookup error. Use TDA's built-in accessors for an episode instead: `ts` and `tf` (start and finish time), `org` and `des` (origin and destination state), `time` (`tf - ts`, and the same built-in `tda_rate`'s `define` can use). A covariate, by contrast, is referenced by its own name from formula, translated automatically the same way `tda_frml` does it.

Value

An object of class `tda_fit`. With `residuals = TRUE`, also carries `$residuals`.

See Also

Other sequence analysis: `tda_evalfi()`, `tda_frml()`, `tda_seq_info()`, `tda_seqgc()`, `tda_seqm()`, `tda_seqmd()`, `tda_seqpe()`, `tda_seqpm()`

Examples

```
set.seed(1)
n <- 200
d <- data.frame(x = rnorm(n))
haz <- exp(0.5 + 0.8 * d$x)
d$dur <- rexp(n, haz)
d$status <- 1
# the exponential hazard model, written out by hand -- matches
# tda_rate(Surv(dur, status) ~ x, d, model = "exponential"). des and
# (tf - ts) are TDA's built-ins, not d's dur/status columns -- see
# Details.
fit <- tda_frml(Surv(dur, status) ~ x, d,
               c("xb = b0 + x * b1", "gamma = exp(xb)",
                 "fn = des * xb - gamma * (tf - ts)"))
coef(fit)
coef(tda_rate(Surv(dur, status) ~ x, d, model = "exponential"))

# the same model, written in R syntax instead
fit2 <- tda_frml(Surv(dur, status) ~ x, d, {
  xb = b0 + x * b1
  gamma = exp(xb)
  fn = des * xb - gamma * (tf - ts)
})
coef(fit2)
```

tda_g	<i>Run a graph command</i>
-------	----------------------------

Description

tda_g reaches any of TDA's forty graph commands; the named functions are the ones with an obvious meaning.

Usage

```
tda_g(g, cmd, options = list(), dir = tempfile("tda"), ...)
```

Arguments

g	a tda_graph .
cmd	the command name, e.g. "gsp", "gmst".
options	a named list of options for the command.
dir	working directory.
...	passed to tda_run .

Value

An object carrying a table and the run.

See Also

Other graph analysis: [plot.tda_graph\(\)](#), [tda_dmet\(\)](#), [tda_g_analyses](#), [tda_graph\(\)](#), [tda_locate_line\(\)](#), [tda_ptree\(\)](#), [tda_ucl\(\)](#)

Examples

```
e <- data.frame(from = c(1, 1, 2, 3), to = c(2, 3, 4, 4), value = 1)
g <- tda_graph(e, directed = FALSE)
tda_g(g, "gmst")$table # minimum spanning tree
```

tda_g_analyses	<i>Graph analyses</i>
----------------	-----------------------

Description

One function per graph command, each a thin wrapper: the command name, the options it takes and the table TDA writes. Everything TDA offers that has no named wrapper is still reachable with [tda_g](#).

Usage

```
tda_g_components(g, edges = FALSE, ...)

tda_g_shortest(g, format = c("reachable", "matrix", "matrix_no_labels"), ...)

tda_g_mst(g, algorithm = c("kruskal", "prim"), max = FALSE, sort = FALSE, ...)

tda_g_cliques(
  g,
  algorithm = c("bron_kerbosch", "harary_ross"),
  min_size = NULL,
  sort = FALSE,
  ...
)

tda_g_cycles(
  g,
  format = c("fundamental_nodes", "fundamental_edges", "all_v1", "all_v2"),
  ...
)

tda_g_cutpoints(g, per_node = FALSE, ...)

tda_g_flow(g, matrix = FALSE, ...)

tda_g_degrees(g)

tda_g_edges(
  g,
  format = c("edges", "lower_triangle", "lower_triangle_diag", "full", "full_square"),
  ...
)

tda_g_transitive(g, ...)

tda_g_aggregate(g, ...)
```

```
tda_g_symmetric(g, ...)

tda_g_toposort(g, relabel_graph = FALSE, ...)

tda_g_nodes(g, ...)

tda_g_dot(g, ...)

tda_g_links(g, backward = FALSE, ...)

tda_g_dcycles(
  g,
  format = c("per_cycle", "per_node", "progress"),
  stop_at_first = FALSE,
  ...
)

tda_g_dblocks(g, ...)

tda_g_compact(g, per_node = FALSE, ...)

tda_g_gcliques(g, ...)

tda_g_random(g, ...)

tda_g_ownership(g, ...)

tda_g_backward(g, ...)

tda_g_paths(g, ...)

tda_g_flowcontrol(g, ...)

tda_g_centred(g, ...)

tda_g_spantrees(g, edges = FALSE, ...)

tda_g_reachable(
  g,
  format = c("count", "list", "strong_nodes", "strong_edges"),
  ...
)

tda_g_neighbourhoods(g, ...)

tda_g_independent(g, min_size = NULL, use_complement = FALSE, ...)

tda_g_spanning(g)
```

```
tda_g_eigen(g, n = NULL, tol = NULL, control = NULL, ...)
```

Arguments

<code>g</code>	a tda_graph .
<code>edges</code>	for <code>tda_g_components</code> , report each component as an edge list instead of a node list (<code>opt=2</code>). For <code>tda_g_spantrees</code> , an edge list instead of one record per tree (<code>gnst's opt=2</code>).
<code>...</code>	options for the command.
<code>format</code>	for <code>tda_g_shortest</code> : "reachable" (default, only reachable pairs), "matrix" (a square matrix with row/ column labels), or "matrix_no_labels" – <code>gsp's opt=</code> . For <code>tda_g_cycles</code> : "fundamental_nodes" (default), "fundamental_edges", "all_v1", or "all_v2" – <code>gcyc's opt=</code> . For <code>tda_g_edges</code> : "edges" (default), or the adjacency matrix as "lower_triangle", "lower_triangle_diag", "full", or "full_square" – <code>gdp's opt=</code> . For <code>tda_g_dcycles</code> : "per_cycle" (default), "per_node", or "progress" (report progress to the console instead of writing anything) – <code>gdcyc's opt=</code> . For <code>tda_g_reachable</code> : "count" (default, number of reachable nodes), "list" (plus the node numbers), "strong_nodes", or "strong_edges" (strongly connected components) – <code>gdcon's opt=</code> . Five different commands' own <code>opt=</code> , each with its own meaning.
<code>algorithm</code>	for <code>tda_g_mst</code> : "kruskal" (default) or "prim". For <code>tda_g_cliques</code> : "bron_kerbosch" (default) or "harary_ross". Different commands' own <code>alg=</code> .
<code>max</code>	for <code>tda_g_mst</code> , find a maximum spanning tree instead of a minimum one (<code>max=1</code>).
<code>sort</code>	for <code>tda_g_mst</code> and <code>tda_g_cliques</code> , sort the output by node number.
<code>min_size</code>	for <code>tda_g_cliques</code> , the minimum clique size (<code>gcliq's min=</code> , default 3). For <code>tda_g_independent</code> , the minimum size of the independent set to search for (<code>giset's min=</code>).
<code>per_node</code>	for <code>tda_g_cutpoints</code> , one record per node instead of one per component (<code>gcut's opt=2</code>). For <code>tda_g_compact</code> , the same idea for <code>gcset's opt=2</code> .
<code>matrix</code>	for <code>tda_g_flow</code> , report the result as a square flow matrix instead of one record per flow (<code>opt=2</code>).
<code>relabel_graph</code>	for <code>tda_g_toposort</code> , return the graph with its new labels instead of the list of old/new labels (<code>gsort's own opt=2</code>).
<code>backward</code>	for <code>tda_g_links</code> , backward links instead of forward ones (<code>gdln's opt=2</code>).
<code>stop_at_first</code>	for <code>tda_g_dcycles</code> , stop after the first cycle is found (<code>gdcyc's dopt=1</code>).
<code>use_complement</code>	for <code>tda_g_independent</code> , search the complementary graph instead (<code>giset's cg=1</code>).
<code>n</code>	for <code>tda_g_eigen</code> , the number of eigenvalues/vectors to compute (<code>gev's n=</code> , default 1).
<code>tol</code>	for <code>tda_g_eigen</code> , the required accuracy (<code>gev's eps=</code>).
<code>control</code>	for <code>tda_g_eigen</code> , convergence settings from tda_control (only its <code>maxit</code> field applies, as <code>gev's mxit=</code>).

Details

The descriptions below are Rohwer's, from TDA's manual; the exact wording and the full option list for any command are available with `tda_help("gmst")` and so on.

Value

An object carrying `table` – the records the command writes, in that command's layout (edges for the tree and path commands, node rows for the centrality and component ones; see the command's `tda_help()` entry) – or, for the few commands whose output is ragged or non-tabular, `table` as a ragged record list or `text` as plain lines.

`tda_g_backward` also carries `summary`, one row per node, from the second file `gbcf` writes; TDA does not produce it unless a file is named for it, so the wrapper asks by default.

Connectivity and structure

`tda_g_components` connected components (`gcon`).

`tda_g_reachable` for each node of a directed graph, the set of nodes reachable from it by a directed path (`gdcon`).

`tda_g_cutpoints` cut nodes of an undirected graph (`gcut`).

`tda_g_transitive` transitive closure (`gtcl`).

`tda_g_symmetric` whether every edge from *i* to *j* is matched by one from *j* to *i* (`gsym`).

`tda_g_toposort` topological sort of a directed graph (`gsort`).

Subgraphs

`tda_g_cliques` all cliques – maximally connected subgraphs – of an undirected graph (`gcliq`).

`tda_g_independent` a maximal independent set, found by a greedy randomized search procedure (GRASP), so the answer is a good one rather than a proven optimum (`giset`).

`tda_g_compact` compact sets in an undirected valued graph (`gcset`).

`tda_g_neighbourhoods` compares the neighbourhoods of the nodes and groups those that agree. Needs an unvalued graph; the wrapper arranges that (`gcni`).

Paths, trees and cycles

`tda_g_shortest` a shortest path for each pair of nodes. Directed or undirected, valued or unvalued; in an unvalued graph every edge counts as 1 (`gsp`).

`tda_g_paths` all paths between any two nodes, with their minimal or maximal length (`gep`).

`tda_g_mst` a minimum, or maximum, spanning tree for each connected component of an undirected valued graph (`gmst`).

`tda_g_spantrees` enumerates every spanning tree of an undirected graph (`gnst`).

`tda_g_cycles` a fundamental set of cycles, and optionally all cycles, of an undirected graph (`gcyc`).

`tda_g_dcycles` cycles in a directed graph (`gdcyc`).

`tda_g_links` forward and backward links in a directed graph (`gdln`).

Flow, control and ownership

tda_g_flow a maximal flow between each pair of nodes joined by a directed path (gflow).
 tda_g_flowcontrol the quantities from which measures of flow control are built, for a directed valued graph (gfc).
 tda_g_ownership integrated ownership in a directed valued graph (gio).
 tda_g_backward direct and indirect backward control (gbcf).
 tda_g_centred node-centred networks (gnc).

Description and output

tda_g_nodes information about the nodes of the current graph (gni).
 tda_g_eigen eigenvalues and eigenvectors of the adjacency matrix of an undirected graph (gev).
 tda_g_aggregate aggregates the nodes of a valued graph (gda).
 tda_g_dblocks blocks of a directed graph (gdcset).
 tda_g_random generates an unvalued graph with a given number of nodes and edges (gcd).
 tda_g_dot, tda_g_edges write the graph out as a Graphviz dot file, or an edge list/adjacency matrix, respectively (gdot, gdp).
 tda_g_degrees node degrees from gni, TDA's node-information command: in-degree, out-degree, loops, and a degree column (for an undirected graph each incident edge is reported on both sides, so the degree is the in-degree, not the sum). Takes no further arguments.

What each command requires

TDA refuses a command whose graph is of the wrong kind rather than answering wrongly, so the error message is worth reading. Most of the gd* commands want a directed graph, gcliq, gcut, gcyc, gmst, gnst, gcset and gnc want an undirected one, and gcni wants an unvalued one.

See Also

Other graph analysis: [plot.tda_graph\(\)](#), [tda_dmet\(\)](#), [tda_g\(\)](#), [tda_graph\(\)](#), [tda_locate_line\(\)](#), [tda_ptree\(\)](#), [tda_ucl\(\)](#)

Examples

```
# a small directed graph: 1 and 2 both feed 3, which feeds 4, which
# feeds 5
e <- data.frame(from = c(1, 1, 2, 2, 3, 4),
                 to   = c(2, 3, 3, 4, 4, 5),
                 value = c(1, 2, 1, 3, 2, 1))

# undirected questions want an undirected graph
g <- tda_graph(e, directed = FALSE)
tda_g_components(g)$table
tda_g_cliques(g)$table
tda_g_mst(g)$table
tda_g_cycles(g)$table
tda_g_cutpoints(g)$table
```

```

tda_g_compact(g)$table
tda_g_spantrees(g)$table
tda_g_spanning(g)$table
tda_g_independent(g)$table
tda_g_centred(g)$table

# directed ones want a directed graph
d <- tda_graph(e, directed = TRUE)
tda_g_toposort(d)$table
tda_g_reachable(d)$table
tda_g_transitive(d)$table
tda_g_nodes(d)$table
tda_g_links(d)$table
tda_g_paths(d)$table
tda_g_flow(d)$table
tda_g_flowcontrol(d)$table
tda_g_backward(d)$table
tda_g_degrees(d)[, c("node", "in_degree", "out_degree")]
tda_g_edges(d)$table      # the raw edge dump (gdp)
tda_g_random(d)$table     # a fresh random graph, ignores e's edges
tda_g_dot(d)$text         # Graphviz source, not a table -- see tda_g()
tda_g_shortest(g)$table   # all-pairs shortest paths

# a denser graph, with an actual triangle and edges valued in [0, 1] --
# what gcliques (a real clique, not just a connected pair) and gio (an
# ownership fraction) both need to find something on this small example
e2 <- data.frame(from = c(1, 2, 3, 1, 4), to = c(2, 3, 1, 4, 5),
                 value = c(0.3, 0.5, 0.8, 0.6, 0.9))
g2 <- tda_graph(e2, directed = FALSE)
d2 <- tda_graph(e2, directed = TRUE)
tda_g_gcliques(g2)$table
tda_g_ownership(d2)$table

# aggregate collapses named nodes into new ones -- rcn=, TDA's
# recode syntax, is required; without it there is nothing to do
tda_g_aggregate(d, rcn = "1[1,2],2[3,4,5]")$table

# neighbourhoods wants an unvalued graph (gcni's restriction)
e3 <- data.frame(from = c(1, 1, 2, 2, 3, 4), to = c(2, 3, 3, 4, 4, 5))
tda_g_neighbourhoods(tda_graph(e3, directed = FALSE))$table

# this graph has no cycles (it is a DAG) and every edge already agrees
# with itself in one direction, so these two are empty rather than
# broken -- not every command has something to report
tda_g_dcycles(d)$table
tda_g_symmetric(d)$run$output[grepl("Nodes|symmetr", d$run$output)]
tda_g_dblocks(d)$table # 0 compact blocks in this graph, also real
tda_g_eigen(g)        # small graphs like these often do not converge

# anything without a named wrapper goes through tda_g()
tda_g(d, "gdp")$table

```

tda_gap	<i>The assignment problem</i>
---------	-------------------------------

Description

TDA's gap on a cost matrix: the permutation that assigns each row to a column at minimal total cost (Carpaneto and Toth's algorithm), the manual's 7.3.1.1. The matrix goes to TDA as a full matrix graph (gdd(opt=7)), so it need not be symmetric.

Usage

```
tda_gap(cost, ...)
```

Arguments

cost	a square numeric cost matrix.
...	passed to tda_run .

Value

assignment, the permutation as a two-column matrix (i, p(i)); permuted, the cost matrix with its rows permuted accordingly; and output.

Examples

```
cost <- rbind(c(60, 0, 0, 76, 0, 0), c(0, 40, 18, 0, 60, 24),
             c(60, 16, 2, 4, 0, 40), c(0, 27, 18, 3, 55, 75),
             c(0, 40, 62, 16, 11, 53), c(28, 4, 10, 84, 0, 16))
tda_gap(cost)$assignment
```

tda_gap_permute	<i>Column permutations toward graph agreement</i>
-----------------	---

Description

TDA's gap (CACM algorithm 548) on two integer-valued graphs, given as matrices the way [tda_gqap](#) takes them.

Usage

```
tda_gap_permute(g1, g2, ...)
```

Arguments

g1, g2	symmetric integer matrices of equal dimension.
...	passed to tda_run .

Value

permuted, the permuted matrix, and output.

Examples

```
f <- rbind(c(0, 3, 1), c(3, 0, 2), c(1, 2, 0))
d <- rbind(c(0, 1, 4), c(1, 0, 2), c(4, 2, 0))
r <- tda_gap_permute(f, d)
r$permuted
```

tda_gde	<i>Undirected graph from node lists</i>
---------	---

Description

TDA's gde: builds an undirected graph from a data matrix where node numbers each unit and id identifies the objects they share; units sharing an id become adjacent.

Usage

```
tda_gde(node, id, output = c("edges", "matrix"), loops = TRUE, ...)
```

Arguments

- node, id integer vectors of equal length.
- output "edges" for an edge list, "matrix" for the square adjacency matrix.
- loops FALSE removes loops.
- ... passed to [tda_run](#).

Value

the values TDA wrote, as a matrix (one row per line of the output file), from the export channel.

Examples

```
# three persons (node) and the organisations they belong to (id):
# person 1 in 10 and 20, person 2 in 20 and 30, person 3 in 30.
# Sharing an organisation makes two persons adjacent: 1-2 and 2-3.
tda_gde(node = c(1, 1, 2, 2, 3), id = c(10, 20, 20, 30, 30))
```

tda_gdf

*General distribution functions for censored data***Description**

TDA's gdf: the marginal (or, with method, joint) distribution or survivor function for possibly right-censored data, via a Kaplan-Meier-style calculation – the same underlying censoring machinery as [tda_lsreg1](#), but estimating the distribution function directly rather than a conditional expectation.

Usage

```
tda_gdf(
  formula,
  data,
  censor = NULL,
  id = NULL,
  dimension = NULL,
  what = c("distribution", "survivor", "expected_values"),
  method = c("marginal", "joint1", "joint2"),
  offset = NULL,
  n_boxes = NULL,
  delta = NULL,
  control = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

formula	a one-sided formula naming the (possibly censored) variable, e.g. $\sim Y$, or a list of such formulas – one per dimension – for the multi-dimensional, grp=-based form (see tda_lsreg1 's Details for how id groups rows into units).
data	a data frame.
censor	the censoring indicator, as a name or a vector, or NULL (the default) when every observation is exact: TRUE or 1 marks a case whose value is censored (the opposite of survival::Surv's event indicator). TDA's cen= is 1 for an exact observation and is translated here.
id	for multi-dimensional data, the name of the column identifying which unit each row belongs to (TDA's grp=).
dimension	with id, the name of the column saying which margin (dimension) each of a unit's observations belongs to – TDA's level variable in grp=ID,L1. Positive integers, not necessarily contiguous; TDA requires that no two observations of one unit share a value, so this is mandatory whenever an id value repeats. The data are in long format: one row per (unit, margin), and the joint methods estimate the joint distribution over the margins.

what	the distribution function (default), the survivor function, or the expected value of each case's observation under the fitted distribution – gdf's prn=. The third option returns a differently-shaped table: one row per case (not per distinct value), carrying the observed value, its exact/censored status (1 = exact, TDA's cen value), and its expected value.
method	"marginal" (default) or, for multi-dimensional data (with id), "joint1"/"joint2" – gdf's own opt=, the same three values tda_lsreg1's method takes.
offset, n_boxes	for method = "marginal" or "joint1" only, the domain offset and the number of grid boxes (gdf's sc=/n=).
delta	for method = "joint2" only, the delta grid spacing (gdf's d=).
control	for method = "marginal" or "joint1" only, convergence settings (only tda_control's maxit/tolf fields apply, as gdf's mxit=/tolf=).
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to tda_run.

Value

An object carrying a table of values and the distribution (or survivor) function at each.

See Also

Other regression: [TDA_FAMILIES](#), [TDA_QRMODELS](#), [tda_freg\(\)](#), [tda_llreg\(\)](#), [tda_lsreg\(\)](#), [tda_mlrc_design\(\)](#), [tda_mreg\(\)](#), [tda_nlreg\(\)](#), [tda_npreg\(\)](#), [tda_zreg\(\)](#)

Examples

```
set.seed(1)
n <- 20
d <- data.frame(y = round(rexp(n, 0.2), 1), cen = rbinom(n, 1, 0.3))
fit <- tda_gdf(~ y, d, censor = "cen")
fit$table

# expected values: one row per case, not per distinct value
fit2 <- tda_gdf(~ y, d, censor = "cen", what = "expected_values")
head(fit2$table)
```

tda_gdu

Union of two graphs

Description

TDA's gdu: graph 1 is loaded from edges1 (with the perm option of gdd), then the union with the edge list edges2 is written to the output file.

Usage

```
tda_gdu(edges1, edges2, missing_value = -1, ...)
```

Arguments

```
edges1, edges2  data frames: from, to, value.
missing_value    substitute written for missing values.
...              passed to tda\_run.
```

Value

the output file's lines.

Examples

```
e1 <- data.frame(i = c(1, 2), j = c(2, 3), v = c(1, 1))
e2 <- data.frame(i = c(1, 3), j = c(3, 1), v = c(2, 2))
tda_gdu(e1, e2)  # value1/value2 side by side, -1 where absent
```

tda_gfcf

Forward control flows in a directed graph

Description

TDA's gfcf on a directed graph loaded from an edge list.

Usage

```
tda_gfcf(edges, threshold = 0.5, variant = 1, ...)
```

Arguments

```
edges           data frame: from, to.
threshold       control threshold, default 0.5.
variant         1 or 2 (the two forms the command documents).
...             passed to tda\_run.
```

Value

one row per node: the node, how many nodes it controls, and the controlled nodes as a comma-joined string; the pair table TDA writes (node, controlled node, path length L, share S, M, R – the manual's Box 2 of 7.6.1.1) is the "pairs" attribute.

Examples

```
# 1 -> 2 -> 4 and 1 -> 3 -> 4: node 1 controls everything downstream,
# 2 and 3 control node 4 only, node 4 controls nothing
fc <- tda_gfcf(data.frame(i = c(1, 1, 2, 3), j = c(2, 3, 4, 4)))
fc
attr(fc, "pairs") # per pair: path length and the flow shares
```

tda_ghd

*Hasse diagram of multivariate patterns***Description**

TDA's ghd: the rows of x are (assumed unique) patterns; the command derives the dominance order's Hasse diagram.

Usage

```
tda_ghd(x, ...)
```

Arguments

x matrix or data frame of patterns.
 $\dots$ passed to [tda_run](#).

Value

the edge list lines of the diagram.

Examples

```
# five patterns over two criteria; a row dominates another when it
# is at least as large in both. The Hasse diagram keeps only the
# covering relations:  $(0,0) < (1,0)$ ,  $(0,1) < (1,1) < (2,1)$ 
tda_ghd(rbind(c(0, 0), c(1, 0), c(0, 1), c(1, 1), c(2, 1)))
```

tda_gpro

*Projection of proximities***Description**

TDA's gpro on a proximity matrix given as an undirected valued graph.

Usage

```
tda_gpro(d, max_iterations = 50, ...)
```

Arguments

`d` symmetric proximity matrix.
`max_iterations` iteration cap.
`...` passed to `tda_run`.

Value

projection, the projected coordinates, and output.

Examples

```
r <- tda_gpro(as.matrix(dist(cbind(c(0, 0, 3, 3), c(0, 2, 0, 2)))))
r$projection
```

tda_gqap	<i>Quadratic assignment (GRASP)</i>
----------	-------------------------------------

Description

TDA's gqap: given a flow graph and a distance graph (both undirected, integer-valued), search for the assignment minimizing total cost, with the GRASP heuristic of CACM algorithm 754.

Usage

```
tda_gqap(flow, dist, max_iterations = 100, ...)
```

Arguments

`flow, dist` symmetric integer matrices of equal dimension.
`max_iterations` iteration cap.
`...` passed to `tda_run`.

Value

with direct exports enabled, a list with cost, assignment and output; otherwise the printed output lines.

Examples

```
f <- rbind(c(0, 3, 1), c(3, 0, 2), c(1, 2, 0))
d <- rbind(c(0, 1, 4), c(1, 0, 2), c(4, 2, 0))
tda_gqap(f, d)
```

tda_graph

*A graph for TDA's graph commands***Description**

A graph for TDA's graph commands

Usage

```
tda_graph(
  edges,
  directed = TRUE,
  form = c("edges", "dissimilarity"),
  options = list()
)
```

Arguments

edges	an edge list – from, to, and one value, or two values when the graph carries a different one in each direction, as the shipped <code>gd1.dat</code> does – or a square adjacency matrix, or a <code>dist</code> .
directed	whether the graph is directed. <code>FALSE</code> emits <code>gt=2</code> . Several commands – tda_g_components among them, and plot.tda_graph 's <code>layout = "tree"</code> – only work on an undirected graph, and TDA itself refuses a directed one for them; <code>directed = TRUE</code> (the default here) is right for ordinary, arrow-drawn graphs, but a tree needs <code>FALSE</code> , since a tree's layout comes from which nodes are connected, not which way an edge happens to point.
form	how TDA is to read it: <code>"edges"</code> for an edge list, which is <code>gdd</code> option 1, or <code>"dissimilarity"</code> for the upper triangle of a dissimilarity matrix, option 4.
options	a named list of further <code>gdd</code> options.

Value

A `tda_graph`.

See Also

Other graph analysis: [plot.tda_graph\(\)](#), [tda_dmet\(\)](#), [tda_g\(\)](#), [tda_g_analyses](#), [tda_locate_line\(\)](#), [tda_ptree\(\)](#), [tda_ucl\(\)](#)

Examples

```
e <- data.frame(from = c(1, 1, 2, 3), to = c(2, 3, 4, 4), value = 1)
# gcon works on an undirected graph only, so `directed = FALSE`
tda_g_components(tda_graph(e, directed = FALSE))
```

tda_gtopo	<i>Read a GTOPO30 elevation profile</i>
-----------	---

Description

TDA's gtopo: reads a raw GTOPO30 digital-elevation tile (16-bit big-endian integers, row-major from the upper-left corner) and extracts a PROFILE – one latitude across a longitude range, or one longitude across a latitude range. A rectangular window is not supported by the command (established from the source: the one-value-plus-range shapes are the only accepted forms).

Usage

```
tda_gtopo(file, rows, cols, upper_left, pixel_size, lon, lat, ...)
```

Arguments

file	the raw DEM file.
rows, cols	tile dimensions.
upper_left	c(x, y) of the upper-left corner.
pixel_size	c(dx, dy).
lon, lat	the profile: one of them a single value, the other a c(from, to) range.
...	passed to tda_run .

Value

the profile records (a numeric matrix under direct exports, the file's lines otherwise).

Examples

```
f <- tempfile(fileext = ".dem") # synthesize a tiny 3x4 tile
con <- file(f, "wb")
writeBin(as.integer(100 + 1:12), con, size = 2, endian = "big")
close(con)
tda_gtopo(f, rows = 3, cols = 4, upper_left = c(10, 50),
          pixel_size = c(0.5, 0.5), lat = 49.5, lon = c(10, 11.5))
```

`tda_help`*TDA's manual, and two notes about the session*

Description

`tda_help` looks a keyword up in `tda.hlp`, the manual that ships with TDA – the same text the standalone program prints for `help=...` With no keyword it lists every topic.

Usage

```
tda_help(topic = "")
```

```
tda_time()
```

```
tda_memory()
```

Arguments

<code>topic</code>	a keyword. TDA appends its wildcard, so a prefix is enough; an empty topic lists every keyword.
--------------------	---

Details

`tda_time` and `tda_memory` report what TDA reports for its `time` and `mem` commands.

Value

The text TDA printed, invisibly; it is also printed.

See Also

Other TDA infrastructure: [`tda\_ccnt\(\)`](#), [`tda\_output\(\)`](#), [`tda\_read\_table\(\)`](#), [`tda\_run\(\)`](#), [`tda\_write\_data\(\)`](#)

Examples

```
tda_help("rate")
tda_time()
tda_memory()
```

tda_icov

*Covariance and correlation of interval-valued variables***Description**

The interval of all covariance (or correlation) values reachable when each observation varies in its interval – the same estimand as [tda_ivariance](#), computed by TDA's `icov` and `icorr` (added to this build; not in Rohwer's TDA). The formula names two interval pairs: $\sim iv(xlo, xhi) + iv(ylo, yhi)$.

Usage

```
tda_icov(
  formula,
  data,
  max_boxes = NULL,
  max_iter = NULL,
  tol_width = NULL,
  tol_fd = NULL,
  tol_fe = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)

tda_icorr(
  formula,
  data,
  max_boxes = NULL,
  max_iter = NULL,
  tol_width = NULL,
  tol_fd = NULL,
  tol_fe = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

`formula` $\sim iv(xlo, xhi) + iv(ylo, yhi)$.

`data` a data frame with the four bound columns.

`max_boxes, max_iter, tol_width, tol_fd, tol_fe`
 search limits and tolerances, as in [tda_ivariance](#).

`options, dir, ...`
 as everywhere.

Details

Naming the same pair twice is legal but means something specific: the two copies vary independently over the intervals, so `tda_icov(~ iv(a, b) + iv(a, b), d)` is not the interval variance (its lower end can be negative) and the self-correlation is not fixed at 1. The functions warn when they see this. For the variance of one interval variable use `tda_ivar`.

Covariance certifies at realistic sizes. Correlation certifies small problems; beyond that the search stops at its limits, returns the best values found, and warns – raise `max_iter`/`max_boxes` or accept the honest inner values.

Value

An object with `$bounds`.

Examples

```
# three observations, both variables interval-valued; small enough
# that both searches certify instantly (verified against a dense
# grid in the package tests)
d <- data.frame(xl = c(1, 4, 6), xu = c(2, 5, 8),
               yl = c(3, 1, 6), yu = c(5, 2, 9))
tda_icov(~ iv(xl, xu) + iv(yl, yu), d,
         max_iter = 2e5, max_boxes = 1e5)
tda_icorr(~ iv(xl, xu) + iv(yl, yu), d,
          max_iter = 2e5, max_boxes = 1e5)

# a realistic size: covariance certifies, correlation warns honestly
w <- tda_interval_wages()
tda_icov(~ iv(school_lo, school_hi) + iv(wage_lo, wage_hi), w,
         max_iter = 2e5, max_boxes = 1e5)
```

 tda_idf |

 Distribution function of an interval-valued variable |

Description

`tda_idf`: the distribution and density function of a continuous interval-valued variable, as bounds at each partition point – an interval variable does not pin down a single distribution, only a range one is consistent with. `tda_iddf` is the same idea for a discrete interval-valued variable (a finite set of possible values, not a continuum), and, unlike `idf`, does take `opt=` for a self-consistent distribution; `idf` takes only `fmt=`.

Usage

```
tda_idf(
  formula,
  data,
  self_consistent = FALSE,
```

```

    control = NULL,
    options = list(),
    dir = tempfile("tda"),
    ...
)

tda_iddf(
  formula,
  data,
  self_consistent = FALSE,
  control = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)

```

Arguments

formula	a one-sided formula naming one interval-valued variable, $\sim iv(lo, hi)$.
data	a data frame.
self_consistent	also compute the self-consistent distribution (opt=2), not just the min/max/mean bounds (opt=1, default) – opt= of both commands. For <code>tda_idf</code> it comes back as <code>\$self_consistent</code> (partition point and distribution function) with <code>\$converged</code> ; the fixed point satisfies the self-consistency equation to the tolerance (checked in the tests against the equation iterated in R).
control	convergence settings (mxit=tolf via tda_control), defaults 50 and 0.001.
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to tda_run .

Value

An object carrying a table of partition points and the distribution function's bounds and mean at each.

See Also

Other interval-valued data: [tda_bounds\(\)](#), [tda_ilsreg\(\)](#), [tda_imean\(\)](#), [tda_imreg\(\)](#), [tda_inpreg\(\)](#), [tda_ivar1\(\)](#), [tda_ivreg\(\)](#), [tda_sddf\(\)](#)

Examples

```

set.seed(1)
n <- 30
d <- data.frame(lo = round(rnorm(n, 5, 2), 1))
d$hi <- d$lo + round(runif(n, 0.5, 2), 1)
fit <- tda_idf(~ iv(lo, hi), d)

```

```

head(fit$table)

# iddf: the discrete counterpart, with the self-consistent option idf
# itself does not have
fit2 <- tda_iddf(~ iv(lo, hi), d, self_consistent = TRUE)
head(fit2$table)

```

tda_ilsreg

Least squares with an interval-valued response

Description

The response is interval-valued, the regressor a point value – unlike [tda_ivreg](#), where both sides are intervals. Both the slope and the intercept come back as bounds; see [tda_bounds](#).

Usage

```

tda_ilsreg(
  formula,
  data,
  yl,
  censor,
  options = list(),
  dir = tempfile("tda"),
  ...
)

```

Arguments

formula	a formula using <code>iv()</code> for the interval-valued response, e.g. <code>iv(ylo, yhi) ~ x</code> .
data	a data frame.
yl, censor	the response and censoring variables TDA requires, as names or vectors.
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to tda_run .

Value

An object of class `tda_fit`.

See Also

Other interval-valued data: [tda_bounds\(\)](#), [tda_idf\(\)](#), [tda_imean\(\)](#), [tda_imreg\(\)](#), [tda_inpreg\(\)](#), [tda_ivar1\(\)](#), [tda_ivreg\(\)](#), [tda_sddf\(\)](#)

Examples

```
set.seed(2)
d <- data.frame(x = round(rnorm(30), 2))
d$ylo <- 2 + 0.5 * d$x + rnorm(30, sd = 0.2)
d$yhi <- d$ylo + round(runif(30, 0.3, 1), 2)
tda_ilsreg(iv(ylo, yhi) ~ x, d)
```

tda_imean

*Statistics of an interval-valued variable***Description**

The interval-valued counterparts of the ordinary summaries: a mean, a variance and a Gini coefficient computed from (lower, upper) pairs rather than from points. Each is reported as bounds, because an interval variable does not have a single value for these either.

Usage

```
tda_imean(formula, data, options = list(), dir = tempfile("tda"), ...)
```

```
tda_ivariance(
  formula,
  data,
  max_boxes = NULL,
  max_iter = NULL,
  tol_width = NULL,
  tol_fd = NULL,
  tol_fe = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

```
tda_igini(
  formula,
  data,
  x = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

formula	a one-sided formula naming one interval-valued variable, $\sim iv(lo, hi)$.
data	a data frame.

options a named list of further TDA options, passed through.
 dir working directory.
 ... passed to [tda_run](#).
 max_boxes, max_iter, tol_width, tol_fd, tol_fe
 for tda_ivariance only, the branch-and-bound search's limits and tolerances
 (nbox=/mxit=/tolbw=/ tolfd=/tolfe=). tda_imean and tda_igini take no
 such options.
 x for tda_igini, the points at which the Lorenz curve is evaluated.

Value

An object carrying the bounds in \$bounds, and the run.

See Also

Other interval-valued data: [tda_bounds\(\)](#), [tda_idf\(\)](#), [tda_ilsreg\(\)](#), [tda_imreg\(\)](#), [tda_inpreg\(\)](#),
[tda_ivar1\(\)](#), [tda_ivreg\(\)](#), [tda_sddf\(\)](#)

Examples

```
d <- data.frame(lo = c(1, 3, 5, 9), hi = c(2, 4, 8, 12))
tda_imean(~ iv(lo, hi), d)
tda_ivariance(~ iv(lo, hi), d)
tda_igini(~ iv(lo, hi), d)
```

 tda_imreg | *Conditional means with interval-valued data* |

Description

imreg: for each interval formed by the induced partition of the regressor's bounds, the mean of the response bounds over the cases covering it.

Usage

```
tda_imreg(formula, data, options = list(), dir = tempfile("tda"), ...)
```

Arguments

formula a formula using iv() for interval-valued variables; tda_imreg needs both sides interval-valued.
 data a data frame.
 options a named list of further TDA options, passed through.
 dir working directory.
 ... passed to [tda_run](#).

Value

An object carrying table, one row per partition boundary.

See Also

Other interval-valued data: [tda_bounds\(\)](#), [tda_idf\(\)](#), [tda_ilsreg\(\)](#), [tda_imean\(\)](#), [tda_inpreg\(\)](#), [tda_ivar1\(\)](#), [tda_ivreg\(\)](#), [tda_sddf\(\)](#)

Examples

```
set.seed(3)
d <- data.frame(xlo = round(rnorm(30), 2))
d$xhi <- d$xlo + round(runif(30, 0.3, 1), 2)
d$ylo <- 2 + 0.5 * d$xlo + rnorm(30, sd = 0.2)
d$yhi <- d$ylo + round(runif(30, 0.3, 1), 2)
head(tda_imreg(iv(ylo, yhi) ~ iv(xlo, xhi), d)$table)
```

tda_independence	<i>Delta-independence of two integer variables</i>
------------------	--

Description

How far two integer-valued variables are from independent: for each subset of y's values, the largest gap between the distribution of x within that subset and the overall distribution of x – TDA's indep. A delta of 0 means x is distributed identically inside and outside the subset.

Usage

```
tda_independence(
  x,
  y,
  data = NULL,
  partition = NULL,
  dir = tempfile("tda"),
  ...
)
```

Arguments

x, y	integer-valued vectors of equal length, or column names in data.
data	optional data frame supplying x and y.
partition	optional increasing integers: the upper ends of consecutive y subsets.
dir	working directory.
...	passed to tda_run .

Details

By default every distinct value of *y* is its subset. With *partition*, *y*'s range is cut into consecutive subsets: each number is the last *y* value of one subset (TDA's *y*=). Values must be integers; TDA works on the integer grid from the smallest to the largest observed value.

Value

A data frame with one row per subset: *subset* (its number), *values* (the *y* values it contains, as a comma-separated string), and *delta*.

See Also

Other descriptive statistics: [tda_brr\(\)](#), [tda_cov\(\)](#), [tda_dstat\(\)](#), [tda_freq\(\)](#), [tda_ineq\(\)](#), [tda_loglin\(\)](#), [tda_quant\(\)](#), [tda_rcorr\(\)](#), [tda_segr\(\)](#), [tda_subm\(\)](#)

Examples

```
d <- data.frame(x = c(1, 2, 3, 4, 1, 2, 3, 4),
               y = c(1, 1, 1, 1, 2, 2, 2, 2))
tda_independence("x", "y", data = d)
```

tda_ineq	<i>Inequality measures</i>
----------	----------------------------

Description

TDA's *ineq*: for each variable, the (weighted) minimum, maximum, mean and standard deviation, the coefficient of variation (standard deviation over mean), and the Gini coefficient – the mean absolute difference between every pair of values, relative to twice the mean, the standard concentration measure for incomes and similar quantities: 0 when every value is equal, approaching 1 as the total concentrates in one case.

Usage

```
tda_ineq(..., options = list(), dir = tempfile("tda"))
```

Arguments

- ... a data frame, a matrix, or vectors.
- options a named list of further TDA options, passed through.
- dir working directory.

Details

ineq **treats every negative value as missing and drops it** – the manual (section 6.4.2) is explicit that only nonnegative values enter, so cases in the result can be smaller than the number of rows given, and a variable that legitimately runs negative is not summarised correctly by this command. Case weights given via a prior *cwt* command (*options*, or the *weights* arguments of the model functions, set one) weight every measure.

Value

An object carrying table: one row per variable, with cases (nonnegative values only), minimum, maximum, mean, sd, vcoeff and gini.

See Also

Other descriptive statistics: [tda_brr\(\)](#), [tda_cov\(\)](#), [tda_dstat\(\)](#), [tda_freq\(\)](#), [tda_independence\(\)](#), [tda_loglin\(\)](#), [tda_quant\(\)](#), [tda_rcorr\(\)](#), [tda_segr\(\)](#), [tda_subm\(\)](#)

Examples

```
set.seed(29)
d <- data.frame(income = round(rlnorm(80, meanlog = 8, sdlog = 0.6)))
tda_ineq(d)$table
```

tda_inpreg

*Non-parametric regression with interval-valued data***Description**

A kernel-smoothed fit evaluated at points you choose, both sides interval-valued.

Usage

```
tda_inpreg(formula, data, x, options = list(), dir = tempfile("tda"), ...)
```

Arguments

formula	a formula using <code>iv()</code> for interval-valued variables.
data	a data frame.
x	the points at which to evaluate the fit.
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to tda_run .

Value

An object carrying table, one row per evaluation point.

See Also

Other interval-valued data: [tda_bounds\(\)](#), [tda_idf\(\)](#), [tda_ilsreg\(\)](#), [tda_imean\(\)](#), [tda_imreg\(\)](#), [tda_ivar1\(\)](#), [tda_ivreg\(\)](#), [tda_sddf\(\)](#)

Examples

```

set.seed(4)
d <- data.frame(xlo = round(rnorm(20), 2))
d$xhi <- d$xlo + round(runif(20, 0.3, 1), 2)
d$ylo <- 2 + 0.5 * d$xlo + rnorm(20, sd = 0.2)
d$yhi <- d$ylo + round(runif(20, 0.3, 1), 2)
tda_inpreg(iv(ylo, yhi) ~ iv(xlo, xhi), d, x = c(-1, 0, 1))$table

```

tda_integrate

*Integrate, minimise, and evaluate a function***Description**

These take a function written in TDA's expression language, with x as the formal argument, and evaluate it inside TDA. They are not given an R function, because the expression is evaluated by TDA's interpreter.

Usage

```
tda_integrate(expr, from, to, options = list(), dir = tempfile("tda"))
```

```

tda_range(
  expr,
  start = NULL,
  use_derivatives = FALSE,
  max_boxes = NULL,
  max_iter = NULL,
  tol_width = NULL,
  tol_fd = NULL,
  tol_fe = NULL,
  options = list(),
  dir = tempfile("tda")
)

```

```
tda_evalf(expr, options = list(), derivatives = 0, dir = tempfile("tda"))
```

```
tda_minimize(expr, options = list(), dir = tempfile("tda"))
```

```

tda_gmin(
  expr,
  start = NULL,
  use_derivatives = FALSE,
  max_boxes = NULL,
  max_iter = NULL,
  tol_width = NULL,
  tol_fd = NULL,
  tol_fe = NULL,

```

```

options = list(),
dir = tempfile("tda")
)

```

Arguments

<code>expr</code>	a single string, e.g. "x*x" or "exp(-x)*sin(x)".
<code>from, to</code>	the limits of integration.
<code>options</code>	a named list of further TDA options, passed through.
<code>dir</code>	working directory.
<code>start</code>	for <code>tda_range</code> and <code>tda_gmin</code> , a starting value for each parameter, as a list: <code>list(2)</code> for one parameter starting at 2 with TDA's default search box (value-1 to value+1); <code>list(c(2, 0, 10))</code> for the same starting value with an explicit box, <code>[0, 10]</code> . TDA's <code>xp=</code> . (Beware writing the raw comma syntax by hand: TDA reads each further number as another parameter's starting value, not a box width for the one before it; the list form here avoids that.)
<code>use_derivatives</code>	for <code>tda_range</code> and <code>tda_gmin</code> , use the function's derivatives during the search (<code>ns=1</code>); TDA's default does not.
<code>max_boxes, max_iter, tol_width, tol_fd, tol_fe</code>	for <code>tda_range</code> and <code>tda_gmin</code> , the box search's limits and tolerances (<code>nbox=/mxit=/tolbw=/tolfd=/tolfe</code>). Not the same search as <code>tda_minimize</code> 's tda_control -style minimiser, so <code>tda_control</code> does not apply to either of these.
<code>derivatives</code>	for <code>tda_evalf</code> , 0 for the value alone (<code>evalf</code>), 1 for the gradient as well (<code>evalf1</code>), 2 for the Hessian too (<code>evalf2</code>); TDA computes them analytically.

Details

`tda_evalf` does not report a value at all: `xf` is a plotting command, drawing the function over the range `options$x` names (default 1 to 10) to a file called `xplot.ps` in the run's directory – read it with `tda_read_ps(x$run, which = "xplot.ps")`, the same way as any other TDA plot.

Value

An object with the numeric result in `value` (`tda_integrate`, `tda_minimize`, `tda_gmin`), the two bounds in `range` (`tda_range`), or nothing structured (`tda_evalf`, see above); `tda_minimize`/`tda_gmin` also carry estimates, the parameter values at the minimum, and the run is always in `run`.

See Also

Other smoothing: [tda_interp\(\)](#), [tda_isotonic\(\)](#), [tda_mat\(\)](#), [tda_sma\(\)](#), [tda_smd\(\)](#), [tda_spl\(\)](#)

Examples

```

tda_integrate("x*x", 0, 1)$value      # 1/3
tda_range("sin(x)")$range              # [-1, 1] over its default domain
tda_minimize("(x - 2)^2", options = list(xp = "0"))$estimates

```

```
# gmin: a box-search-based *global* minimum, not tda_minimize's
# local search -- its start=/tolerances, not tda_control(). Its
# interval arithmetic does not accept ^, unlike tda_minimize's
# expression -- (x-2)*(x-2), not (x-2)^2
tda_gmin("(x-2)*(x-2)", start = list(c(2, 0, 4)))$value

# evalf: the function value at a point, 1.1342
tda_evalf("sin(x) + x^2", options = list(x = 0.7))$value
```

tda_interp	<i>Interpolate a surface over a grid</i>
------------	--

Description

Interpolate a surface over a grid

Usage

```
tda_interp(x, y, z, rx, ry, options = list(), dir = tempfile("tda"))
```

Arguments

x, y, z	the surface: a point cloud in three variables.
rx, ry	the grids to interpolate over.
options	a named list of further TDA options, passed through.
dir	working directory.

Value

An object carrying a table: one row per grid point, x, y and the interpolated z.

See Also

Other smoothing: [tda_integrate\(\)](#), [tda_isotonic\(\)](#), [tda_mat\(\)](#), [tda_sma\(\)](#), [tda_smd\(\)](#), [tda_spl\(\)](#)

Examples

```
# a plane z = x + 2y, evaluated back on its own grid
g <- expand.grid(x = 1:5, y = 1:5)
g$z <- g$x + 2 * g$y
head(tda_interp(g$x, g$y, g$z, rx = 1:5, ry = 1:5)$table)
```

tda_interval_wages	<i>Artificial interval-valued wage data</i>
--------------------	---

Description

A small, reproducible dataset built for trying the whole interval family on one realistic shape: $n = 40$ people, schooling asked and answered in whole years (a point value), wages reported in brackets that widen with income, with a top-coded highest bracket – the classic survey mix of exact and interval-valued variables. Because the regressor is exact, `tda_ivreg`'s exact method certifies sharp slope bounds instantly here; widen the schooling values into intervals yourself to watch certification get hard.

Usage

```
tda_interval_wages()
```

Details

The generating truth is $\log(\text{wage}) = 0.8 + 0.11 \text{ school} + e$, which produces a wage-level slope around 1.6 over this range; any interval method's answer should be judged against the data's coarseness, not against that number.

Works across the family: `tda_imean`, `tda_ivariance` (with `$sd`), `tda_icov`, `tda_icorr` (which honestly warns that it cannot certify at this size), `tda_ivreg` and `tda_ilsreg`.

Value

A data frame with columns `wage_lo`, `wage_hi`, `school_lo`, `school_hi`.

Examples

```
d <- tda_interval_wages()
tda_imean(~ iv(wage_lo, wage_hi), d)

tda_ivreg(iv(wage_lo, wage_hi) ~ iv(school_lo, school_hi), d,
          method = "exact")
```

tda_isd	<i>Standard deviation of an interval-valued variable</i>
---------	--

Description

The square root of `tda_ivariance`'s bounds – valid because the square root is monotone. Its own function because a variance function printing standard deviations surprised people, reasonably.

Usage

```
tda_isd(formula, data, ...)
```

Arguments

```
formula, data, ...
          as in tda\_ivariance.
```

Value

A named vector, the lower and upper standard deviation.

Examples

```
d <- data.frame(lo = c(1, 3, 5, 9), hi = c(2, 4, 8, 12))
tda_isd(~ iv(lo, hi), d)
```

tda_isotonic	<i>Isotonic regression, with or without tie groups</i>
--------------	--

Description

The non-decreasing sequence closest (least squares) to y – TDA's mmp, and with tie groups its mmp1/mmp2. Without groups this is plain pool-adjacent-violators and agrees with `stats::isoreg` exactly.

Usage

```
tda_isotonic(
  y,
  groups = NULL,
  ties = c("primary", "secondary"),
  dir = tempfile("tda"),
  ...
)
```

Arguments

<code>y</code>	a numeric vector, in the order to be respected.
<code>groups</code>	optional vector of the same length marking tie groups by consecutive runs of equal values.
<code>ties</code>	how tie groups are treated: "primary" or "secondary"; ignored without groups.
<code>dir</code>	working directory.
<code>...</code>	passed to tda_run .

Details

groups marks observations whose order among themselves is not meaningful (ties in the ordering variable). Consecutive equal values form one group – the grouping follows runs, exactly as TDA reads its tie matrix, so `c(1, 1, 2)` is two groups but `c(1, 2, 1)` is three. How a group is treated is ties: "primary" (Kruskal's primary approach, `mmp1`) lets the fitted values untie freely – the group's values are sorted, fitted, and each observation receives its rank slot's value, so the result is monotone only up to reordering within groups; "secondary" (`mmp2`) forces one common value per group – the group mean is fitted and broadcast back.

Value

The fitted values, a numeric vector like `y`.

See Also

Other smoothing: [tda_integrate\(\)](#), [tda_interp\(\)](#), [tda_mat\(\)](#), [tda_sma\(\)](#), [tda_smd\(\)](#), [tda_spl\(\)](#)

Examples

```
tda_isotonic(c(5, 3, 4, 6, 2, 7))
# rows 1-2 tied, rows 4-6 tied:
tda_isotonic(c(5, 3, 4, 6, 2, 7), groups = c(1, 1, 2, 3, 3, 3),
             ties = "secondary")
```

 tda_ivar | *Variance of an interval-valued variable (TDA's ivar)* |

Description

Identical to [tda_ivariance](#): TDA's `ivar` branch-and-bound with tunable limits. This is the command-style name; [tda_ivar1](#) wraps the newer `ivar1` algorithm, and all three compute the same variance bounds.

Usage

```
tda_ivar(
  formula,
  data,
  max_boxes = NULL,
  max_iter = NULL,
  tol_width = NULL,
  tol_fd = NULL,
  tol_fe = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

formula, data, max_boxes, max_iter, tol_width, tol_fd, tol_fe, options,
 dir, ...
 exactly as in [tda_ivariance](#).

Value

See [tda_ivariance](#).

Examples

```
d <- data.frame(lo = c(1, 3, 5, 9), hi = c(2, 4, 8, 12))
tda_ivar(~ iv(lo, hi), d)
```

tda_ivar1	<i>Variance of an interval-valued variable (TDA's ivar1)</i>
-----------	--

Description

Wraps TDA's `ivar1`, the newer, options-less variance algorithm. [tda_ivar](#) wraps the older tunable `ivar` branch-and-bound; both compute the same variance bounds, and the names now follow the commands.

Usage

```
tda_ivar1(formula, data, options = list(), dir = tempfile("tda"), ...)
```

Arguments

formula	a one-sided formula naming one interval-valued variable, <code>~ iv(lo, hi)</code> .
data	a data frame.
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to tda_run .

Value

An object of class `tda_fit`; see [tda_bounds](#) for the variance bounds.

See Also

Other interval-valued data: [tda_bounds\(\)](#), [tda_idf\(\)](#), [tda_ilsreg\(\)](#), [tda_imean\(\)](#), [tda_imreg\(\)](#), [tda_inpreg\(\)](#), [tda_ivreg\(\)](#), [tda_sddf\(\)](#)

Examples

```
d <- data.frame(lo = c(1, 3, 5, 9), hi = c(2, 4, 8, 12))
tda_ivar1(~ iv(lo, hi), d)
```

tda_ivreg

*Interval regression: a coefficient identified only up to bounds***Description**

tda_ivreg, tda_ivreg1 and tda_ivreg2 are TDA's three variants of interval regression – both response and regressor are interval-valued – and tda_ivls solves the same kind of problem as a system of linear interval equations. All four return bounds on beta rather than a point estimate, so `coef()` would be misleading; use [tda_bounds](#).

Usage

```
tda_ivreg(
  formula,
  data,
  method = c("two_step", "heuristic", "exact"),
  max_boxes = NULL,
  max_iter = NULL,
  tol_width = NULL,
  tol_beta = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)

tda_ivreg1(
  formula,
  data,
  search_box = list(alpha = c(-10, 10), alpha_radius = c(0, 5), beta = c(-10, 10),
    beta_radius = c(0, 5)),
  max_boxes = 4000,
  max_iter = 4000,
  tol_width = NULL,
  tol_beta = NULL,
  tol_min = 0.001,
  options = list(),
  dir = tempfile("tda"),
  ...
)

tda_ivreg2(
  formula,
  data,
  method = c("direct", "search", "minimizer", "contour"),
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

```
)

tda_ivls(formula, data, options = list(), dir = tempfile("tda"), ...)
```

Arguments

formula	a formula using <code>iv()</code> for interval-valued variables.
data	a data frame.
method	for <code>tda_ivreg</code> : "two_step" (default, only the first two steps), "heuristic", or "exact" optimization – <code>ivreg</code> 's <code>opt=</code> . For <code>tda_ivreg2</code> : "direct" (default, direct calculation), "search" (direct search), "minimizer" (TDA's general minimiser), or "contour" – the same <code>opt=</code> option with a different meaning for each command. The choice changes the fit: on the same data, "exact" converges to a different beta than "two_step" and can exhaust the box search where the default does not.
max_boxes, max_iter, tol_width, tol_beta	for <code>tda_ivreg</code> and <code>tda_ivreg1</code> , the box search's iteration limits and tolerances (<code>nbox=/mxit=/tolbw=/tolfe=</code>); tda_control does not apply here, these are specific to the interval-regression box search, not TDA's general minimiser. For <code>tda_ivreg1</code> the limits default to 4000 each – TDA's 100/100 rarely completes a real search.
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to tda_run .
search_box	for <code>tda_ivreg1</code> , the domain the box search covers: a named list of <code>c(lower, upper)</code> ranges for <code>alpha</code> , <code>alpha_radius</code> , <code>beta</code> and <code>beta_radius</code> (equivalently a 4x2 matrix, or 8 numbers, four lower then four upper). The solution must lie inside it; widen it if the fitted bounds hug an edge. The radii cannot go below 0.
tol_min	for <code>tda_ivreg1</code> , the certification tolerance (<code>tolfe=</code>): a box is accepted when its inclusion lower bound is within this of the best function value found. See Details for why TDA's default is unusable.

Details

Interval-valued data is a lower and an upper bound per case, a different thing from interval *censoring*, and has no equivalent in base R or in **survival**. `iv(lo, hi)` marks such a variable in a formula, the way `Surv()` marks an episode.

`ivreg1` and `ivreg2` are global-optimization (branch-and-bound) searches, and each reports its bounds in its own format – `ivreg2` a centre and a half-width ("Beta (center-radius): C R", read as `[C - R, C + R]`), and `ivls` a bare "Final bounds" with no label at all.

`ivreg1` needs a word of its own. As shipped, the command's search domain is hard-coded – `alpha` in `[1, 2.5]`, `beta` in `[0, 0.1]`, evidently the dataset it was being developed against – so for almost any other data the answer is simply not inside the searched box and TDA reports "0 finally accepted boxes". The command was left mid-study, not broken: on data whose solution lies inside the box, the search certifies cleanly. `tda_ivreg1` therefore passes the domain in through the package's `sbox=` option (a guarded addition to the C; the standalone program is unchanged), always requests

the `prot=` protocol file, and reads the accepted boxes back: `boxes` holds every box the protocol reports, parameters the hull of the near-optimal ones over (`alpha`, `alpha` radius, `beta`, `beta` radius), and `beta/alpha` – what `tda_bounds` returns – the centre rows of that hull. Certification follows TDA’s rule: a box counts when its inclusion lower bound is within `tol_min` of the best value found. The shipped default of $1e-10$ for that tolerance is unreachable (interval arithmetic over-estimates a box’s range), which is the second reason the command always seemed to fail; `tol_min` defaults to $1e-3$ here instead.

Value

An object of class `tda_fit`.

See Also

Other interval-valued data: `tda_bounds()`, `tda_idf()`, `tda_ilsreg()`, `tda_imean()`, `tda_imreg()`, `tda_inpreg()`, `tda_ivar1()`, `tda_sddf()`

Examples

```
set.seed(1)
d <- data.frame(xlo = 1:20)
d$xhi <- d$xlo + 2
d$ylo <- 2 + 0.5 * d$xlo + rnorm(20)
d$yhi <- d$ylo + 1.5
tda_ivreg(iv(ylo, yhi) ~ iv(xlo, xhi), d)

# ivreg2: converges readily even where ivreg1 often does not
d2 <- data.frame(xlo = c(1, 2, 3, 4))
d2$xhi <- d2$xlo + 1
d2$ylo <- 2 + 0.5 * d2$xlo
d2$yhi <- d2$ylo + 1
tda_ivreg2(iv(ylo, yhi) ~ iv(xlo, xhi), d2)

# ivls: the same interval-equations idea, its bound format
tda_ivls(iv(ylo, yhi) ~ iv(xlo, xhi), d2)
```

 tda_l1reg | *Least absolute deviations regression* |

Description

L1-norm regression, which fits the conditional median rather than the mean and is therefore robust to outliers in the response. `quantreg::rq(tau = 0.5)` is the R analogue.

Usage

```
tda_l1reg(
  formula,
  data,
  intercept = NULL,
  options = list(),
  residuals = FALSE,
  dir = tempfile("tda"),
  ...
)
```

Arguments

formula	a two-sided formula.
data	a data frame.
intercept	whether to include an intercept. Defaults to whatever the formula itself says ($y \sim 0 + x$ or $y \sim x - 1$ already mean no intercept, the ordinary R way, and are honoured); set explicitly to override that.
options	a named list of further TDA options, passed through.
residuals	also return each case's residual, as a residuals component; l1reg's pres=.
dir	working directory.
...	passed to tda_run .

Value

An object of class `tda_fit`.

See Also

Other regression: [TDA_FAMILIES](#), [TDA_QRMODELS](#), [tda_freg\(\)](#), [tda_gdf\(\)](#), [tda_lsreg\(\)](#), [tda_mlrc_design\(\)](#), [tda_mreg\(\)](#), [tda_nlreg\(\)](#), [tda_npreg\(\)](#), [tda_zreg\(\)](#)

Examples

```
set.seed(22)
d <- data.frame(x = round(rnorm(50), 3))
d$y <- round(3 - 0.8 * d$x + rnorm(50, sd = 0.4), 3)
# three outliers: l1reg, fitting the median, should barely notice them
d$y[c(2, 10, 30)] <- d$y[c(2, 10, 30)] + c(15, -12, 20)
coef(tda_l1reg(y ~ x, d))
coef(lm(y ~ x, d)) # pulled toward the outliers
fit <- tda_l1reg(y ~ x, d, residuals = TRUE)
head(fitted(fit))    # equivalent to predict(fit); no newdata needed
head(residuals(fit))
```

tda_locate_line	<i>One-dimensional multifacility location</i>
-----------------	---

Description

Place k new (variable) points among m existing (fixed) points on a line so that the weighted interaction costs are minimal – TDA's `gloc`. Each variable point ends up at one of the fixed points; only the order of the fixed points on the line matters, not their spacing (verified against exhaustive search in the test suite).

Usage

```
tda_locate_line(
  fixed_weights,
  variable_weights = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

<code>fixed_weights</code>	a $k \times m$ matrix: interaction weight between each variable point (row) and each fixed point (column, in line order).
<code>variable_weights</code>	optional symmetric $k \times k$ matrix of weights between the variable points themselves (default: none).
<code>options</code>	a named list of further TDA options for <code>gloc</code> .
<code>dir</code>	working directory.
<code>...</code>	passed to tda_run . The graph-setup options <code>opt=</code> , <code>gt=</code> , <code>perm=</code> and <code>sc=</code> pass through to the underlying edge load, as in tda_g .

Value

An integer vector of length k : for each variable point, the index of the fixed point it is placed at.

See Also

Other graph analysis: [plot.tda_graph\(\)](#), [tda_dmet\(\)](#), [tda_g\(\)](#), [tda_g_analyses](#), [tda_graph\(\)](#), [tda_ptree\(\)](#), [tda_ucl\(\)](#)

Examples

```
# two facilities, four sites; facility 1 mostly serves site 2,
# facility 2 site 3, and the two facilities interact
vf <- rbind(c(8, 1, 1, 2), c(1, 1, 7, 3))
tda_locate_line(vf, variable_weights = rbind(c(0, 5), c(5, 0)))
```

tda_loglin

*Log-linear models of a contingency table***Description**

Builds the many-way contingency table TDA's loglin works from out of the variables named in formula, and fits a log-linear model to it. `coef()`, `vcov()`, and `residuals()` work directly on the result, the same as for any other model in this package.

Usage

```
tda_loglin(
  formula,
  data,
  weights = NULL,
  scale = NULL,
  residuals = FALSE,
  control = NULL,
  options = list(),
  dir = tempfile("tda")
)

## S3 method for class 'tda_loglin'
coef(object, ...)

## S3 method for class 'tda_loglin'
vcov(object, ...)

## S3 method for class 'tda_loglin'
residuals(object, ...)

## S3 method for class 'tda_loglin'
print(x, ...)
```

Arguments

formula	a formula (or a list of formulas) naming the table's dimensions and the model(s) to fit against it – see Details.
data	a data frame.
weights, scale	optional case weights and a scale variable (loglin's <code>w=/scale=</code>). Either a column name in data – removed from the table's dimensions automatically, so it never has to be excluded by hand – or a plain vector as long as data.
residuals	also compute each cell's observed count, fitted value, residual, and scale factor – loglin's <code>pres=</code> . In <code>fit\$residuals</code> (or <code>fit\$models[["..."]]</code> \$residuals for more than one formula), a data frame with one row per table cell.

control	convergence settings (mxit=/tolf= only, loglin's iteration control; the other tda_control fields do not apply here and are rejected by TDA if given).
options	a named list of further TDA options, passed through.
dir	working directory.
object, x	a tda_loglin fit.
...	unused; present for S3 consistency.

Details

TDA's `mod=` syntax refers to a table's dimensions by position – the first variable given is A, the second B, and so on – never by the variable's name, and letters run together ("AB") name the saturated interaction between them (section 6.19.2 of the manual: "AB" expands to "A+B+A.B"). This is built from an ordinary R formula instead – $\sim X1 + X2$ (main effects only), $\sim X1 * X2$ (main effects and their interaction), $\sim X1:X2$ (the interaction alone) – translated to TDA's letters automatically via [terms](#), so nothing about that lettering has to be worked out or written by hand.

$\sim X1 + X2$ fits 5 coefficients, not 2, for the same reason any other model in this package with a categorical predictor does: TDA dummy-codes each level of a variable past its first (with 3-level $X1:X2$: 1 constant + 2 dummies for $X1 + 2$ for $X2 = 5$), not one coefficient per variable the way a table with only two dimensions might suggest.

Give several formulas in a list to fit several models against the same table in one call, the way TDA's repeated `mod=` does (each is a separate model with its own full statistic block, not a combined fit) – TDA's two shipped examples (`l11.cf/l12.cf`) each fit at most one model, so this is the less common case, not the primary interface: `coef()/vcov()/residuals()` only work directly when exactly one formula was given, and name every model by its own deparsed formula when more than one was (`fit$models[["~a * b"]]`).

`loglin` itself never prints a log-likelihood, only the likelihood ratio statistic comparing the fitted model against the saturated one – in `fit$lr/fit$lr_p(fit$models[["..."]])$lr` for more than one formula), not `logLik()`, which has no TDA-reported value to return here.

Value

An object carrying table (the contingency table, its own columns named after formula's real variables and weights – or, with no weights at all, F, the manual's name for a plain count column – never TDA's internal letters or a generic name) and, when exactly one formula was given, coefficients (a data frame, at TDA's full precision, parameter names translated back from A/B/... to formula's variable names), `vcov`, `lr/lr_p/pearson/pearson_p/ f/df`, and, with `residuals = TRUE`, residuals – reachable directly, or through `coef()/vcov()/ residuals()`. With more than one formula, all of this lives in models instead, a list named after each formula given, one entry per model with the same fields.

See Also

Other descriptive statistics: [tda_brr\(\)](#), [tda_cov\(\)](#), [tda_dstat\(\)](#), [tda_freq\(\)](#), [tda_independence\(\)](#), [tda_ineq\(\)](#), [tda_quant\(\)](#), [tda_rcorr\(\)](#), [tda_segr\(\)](#), [tda_subm\(\)](#)

Examples

```
set.seed(1)
n <- 200
d <- data.frame(a = rbinom(n, 1, 0.5) + 1, b = rbinom(n, 1, 0.5) + 1)
fit <- tda_loglin(~a * b, d)
coef(fit)
fit$table

# weighted, with a tighter iteration cap -- weights as a column name
d$w <- runif(n, 0.5, 2)
tda_loglin(~a * b, d, weights = "w",
           control = tda_control(maxit = 30))$lr

# several models against the same table at once, the less common case
cmp <- tda_loglin(list(~a, ~a * b), d)
cmp$models[["~a"]][c("lr", "df")] # main effect of a only
cmp$models[["~a * b"]][c("lr", "df")] # saturated: perfect fit
```

tda_lsreg	<i>Least squares regression</i>
-----------	---------------------------------

Description

tda_lsreg is ordinary least squares, TDA's lsreg. tda_lsreg1 is the same method with a censored dependent variable, TDA's lsreg1: the response is only known to lie beyond its recorded value for the cases the censoring variable marks.

Usage

```
tda_lsreg(
  formula,
  data,
  weights = NULL,
  intercept = NULL,
  robust = FALSE,
  equality = NULL,
  inequality = NULL,
  dgroup = NULL,
  options = list(),
  residuals = FALSE,
  dir = tempfile("tda"),
  ...
)

tda_lsreg1(
  formula,
  data,
  censor,
```

```

    id = NULL,
    method = c("marginal", "joint1", "joint2"),
    intercept = NULL,
    offset = NULL,
    n_boxes = NULL,
    delta = NULL,
    residuals = FALSE,
    control = NULL,
    options = list(),
    dir = tempfile("tda"),
    ...
)

```

Arguments

formula	a two-sided formula, or, for multiple equations fit jointly, an (optionally named) list of them, as described above.
data	a data frame.
weights	optional case weights, as in <code>lm</code> . Not available together with multiple equations.
intercept	whether to include an intercept. Defaults to whatever the formula itself says ($y \sim 0 + x$ or $y \sim x - 1$ already mean no intercept, the ordinary R way, and are honoured); set explicitly to override that.
robust	for <code>tda_lsreg</code> , use a heteroskedasticity-robust covariance matrix instead of the ordinary one; <code>lsreg</code> 's <code>s=1</code> .
equality, inequality	for <code>tda_lsreg</code> , linear constraints on the coefficients, written in the predictors' own names – equality = " $x_1 + x_2 = 6$ " constrains those two coefficients to sum to exactly 6 (<code>lsecon=</code>); inequality is the same syntax but a lower bound, not an exact value (<code>lsicon=</code>). Each can be a character vector for more than one constraint.
dgroup	for <code>tda_lsreg</code> , a categorical variable (a factor, or a character/numeric vector, one value per case) to estimate every level of at once, rather than the usual approach of dropping one level as an arbitrary reference – <code>lsreg</code> 's <code>dgrp=</code> . Also accepts a character vector naming existing one-hot indicator columns in data (region membership already split into separate NE/NC/SO/WE 0/1 columns, say): <code>dgroup = c("NE", "NC", "SO", "WE")</code> reconstructs the per-case group label from them directly, rather than requiring <code>data[c("NE", "NC", "SO", "WE")][max.col(...)]</code> -style setup first. Every case must have exactly one of the named columns set; this is checked explicitly rather than resolved silently. For more than one independent grouping at once (region <i>and</i> sex, say, each with its weighted-sum-to-zero constraint), give a list of these – <code>list(region = c("NE", "NC", "SO", "WE"), sex = c("M", "F"))</code> , or an unnamed list of vectors/factors; a plain, non-list <code>dgroup</code> is always exactly one group. The fit keeps the intercept as the levels' own weighted grand mean and returns one coefficient per level (named <code>dgroup</code> plus the level, e.g. <code>dgroupA</code> for one group, <code>dgroupregionNE/dgroupsexM</code> style for more than one), each that level's deviation from it, under the weighted-sum-to-zero constraint TDA imposes automatically, separately for each group.

options	a named list of further TDA options, passed through: lsecon and lsicon for constraints, dgrp for groups, and the output files.
residuals	for tda_lsreg, ask TDA to write residuals (pres=) so residuals has something to return; off by default, since it costs an extra file TDA has to write whether or not it is used.
dir	working directory.
...	passed to tda_run.
censor	for tda_lsreg1, the censoring indicator, as a name or a vector: TRUE or 1 marks a case whose response is censored, as in survival::Surv. TDA's cen= runs the other way – it marks the cases that are exact – so this is translated for you.
id	for multiple equations, the name of the column identifying which case each row of data belongs to.
method	for multiple equations, how the censored expectations are estimated: "marginal" (default, based on the marginal Kaplan-Meier), or "joint1"/"joint2", two different methods based on the joint distribution across equations – lsreg1's opt=. Meaningless (and rejected) for a single equation, where grp= – which opt=2/3 require – is not used at all. "joint2" refuses any case censored on more than one equation at once.
offset, n_boxes	for tda_lsreg1 with method = "marginal" or "joint1", the box search's domain offset and number of grid boxes (sc=n=).
delta	for tda_lsreg1 with method = "joint2", the delta grid spacing (d=).
control	for tda_lsreg1 with method = "marginal" or "joint1", convergence settings from tda_control: its mxit field reaches the method's general mxit=, and its tolf field reaches the method-specific tolp= (a different tolerance from tda_gdf's tolf=, despite the shared field name in tda_control – lsreg1 has both a general mxit=/ tolp= pair and a method-1-specific mxit1=/tolp= pair; only the general pair is reachable through control here).

Details

Both return a tda_fit, so coef and summary work on them as they do for the model functions. lsreg reports a standard error, a coefficient-to-error ratio and a significance level beside each estimate; where it shows --- it had nothing to compute them from, which happens when the fit is exact and the residual variance is zero. lsreg1 does not print its estimates – it writes them to the file named by ppar= – so tda_lsreg1 always asks for that file and reads it back.

lsreg1 estimates the marginal distribution function of the response before fitting, which is why it iterates; with no censored case it reproduces tda_lsreg exactly, and the estimates move away from the least squares ones as the censored share grows.

tda_lsreg1 also fits several censored regression equations jointly (a system of seemingly-unrelated regressions, SUR) when formula is a list, one formula per equation, and id names the column identifying which case each row belongs to – TDA's grp= option. Each equation gets its own 0/1 indicator and its own zero-padded copy of every predictor (zero outside its equation's rows), built here automatically; this standard SUR construction is what gives each equation its own, separate coefficients, rather than one fit pooled across all of them.

Value

An object of class `tda_fit`. With `dgroup=` it also carries groups: one row per group, its case count and its weight, which TDA prints and stores nowhere else. `summary()` shows a p-value column computed from the coefficient's test statistic with the same reference distribution TDA itself uses (a t on the residual degrees of freedom here), alongside the residual standard error, R-squared, and the F statistic with its p-value. Cases with a missing value in the response or any predictor are dropped before TDA sees the data, with a message (see `?tdaR` on missing values).

Experimental

`tda_lsreg1` is experimental. TDA's `lsreg1` does not print a standard error, a C/Error ratio, a log likelihood or a convergence flag the way the model commands do, so `coef()` works but `summary()`, `logLik()` and `vcov()` carry NA or NULL where the other fitting functions carry a value. This reflects what TDA itself reports, not a parsing gap. `predict()` and `fitted()` are not available for the multi-equation form – `coef()` and `summary()` are.

See Also

Other regression: [TDA_FAMILIES](#), [TDA_QRMODELS](#), [tda_freg\(\)](#), [tda_gdf\(\)](#), [tda_llreg\(\)](#), [tda_mlrc_design\(\)](#), [tda_mreg\(\)](#), [tda_nlreg\(\)](#), [tda_npreg\(\)](#), [tda_zreg\(\)](#)

Examples

```
set.seed(1)
d <- data.frame(x = 1:40, z = rnorm(40))
# with an error term: fitting a response that is an exact function of the
# predictors leaves no residual variance, and the standard errors come
# back as "---" because there is nothing to divide by
d$y <- 2 + 0.5 * d$x + d$z + rnorm(40)
fit <- tda_lsreg(y ~ x + z, d)
coef(fit)
summary(fit)

# a constrained fit: the two coefficients forced to sum to exactly 8
# (matching the true generating values, 3 and 5), a real feature of
# lsreg itself (lsecon=), not a general optimiser option --
# coefficients are written in terms of x1 and x2, not TDA's
# b1/b2 labels
d2 <- data.frame(x1 = rnorm(60), x2 = rnorm(60))
d2$y <- 3 * d2$x1 + 5 * d2$x2 + rnorm(60, sd = 0.5)
cfits <- tda_lsreg(y ~ x1 + x2, d2, intercept = FALSE,
                  equality = "x1 + x2 = 8")
coef(cfits)
sum(coef(cfits)) # exactly 8, by construction

# dgroup: every level of a categorical predictor estimated at once,
# not one dropped as a reference -- true group effects are 1, 3, 5
set.seed(2)
n <- 120
g <- sample(c("A", "B", "C"), n, replace = TRUE)
d3 <- data.frame(x = rnorm(n))
```

```

eff <- c(A = 1, B = 3, C = 5)[g]
d3$y <- eff + 0.5 * d3$x + rnorm(n, sd = 0.3)
gfit <- tda_lsreg(y ~ x, d3, dgroup = g)
coef(gfit)  # intercept is the weighted grand mean; each dgroup
            # coefficient its level's deviation from it

# fitted values and residuals: predict() with no newdata falls back to
# the data the model was fit on; residuals() needs residuals = TRUE at
# fit time (TDA has to be asked to write them, so it is not automatic)
fit2 <- tda_lsreg(y ~ x + z, d, residuals = TRUE)
head(predict(fit2))
head(residuals(fit2))
predict(fit2, newdata = data.frame(x = c(0, 20), z = c(0, 0)))

# the same data with the response censored above 22: only the cases
# below it are observed, the rest are known only to lie beyond
d$cen <- as.integer(d$y > 22)
d$yc <- pmin(d$y, 22)
coef(tda_lsreg1(yc ~ x + z, d, censor = "cen"))

# two censored equations, fit jointly: true coefficients are 2, 0.8
# for the first and 5, -0.5 for the second
set.seed(2)
n <- 60
d2 <- data.frame(id = 1:n, x1 = rnorm(n), x2 = rnorm(n))
y1 <- 2 + 0.8 * d2$x1 + rnorm(n, sd = 0.4)
y2 <- 5 - 0.5 * d2$x2 + rnorm(n, sd = 0.4)
d2$cen1 <- as.integer(y1 > 2.5)
d2$cen2 <- as.integer(y2 > 5.5)
d2$y1 <- pmin(y1, 2.5)
d2$y2 <- pmin(y2, 5.5)
sur <- tda_lsreg1(list(Eq1 = y1 ~ x1, Eq2 = y2 ~ x2), d2,
                  censor = list("cen1", "cen2"), id = "id")
coef(sur)
coef(tda_lsreg1(list(Eq1 = y1 ~ x1, Eq2 = y2 ~ x2), d2,
                  censor = list("cen1", "cen2"), id = "id",
                  method = "joint1")) # a different fit, not the same
                                     # numbers under a new name

```

tda_mag

Aggregating a matrix by row and column groups

Description

Sums A 's entries within each (row-group, column-group) block – TDA's `mag`. `row_groups/col_groups` assign each row/column of A to a group (any positive integers; groups need not be contiguous or start at 1). The result has one row per distinct value in `row_groups` and one column per distinct value in `col_groups`, in ascending order of group number.

Usage

```
tda_mag(A, row_groups, col_groups, dir = tempfile("tda"))
```

Arguments

A a numeric matrix.
row_groups integer group index for each row, length nrow(A).
col_groups integer group index for each column, length ncol(A).
dir working directory.

Value

A numeric matrix.

See Also

Other matrix reshaping: [tda_mcat\(\)](#), [tda_mcvec\(\)](#), [tda_mrsum\(\)](#), [tda_msort\(\)](#), [tda_msrow\(\)](#), [tda_mtrim\(\)](#)

Examples

```
A <- matrix(1:12, 3, 4, byrow = TRUE)
tda_mag(A, row_groups = c(1, 2, 1), col_groups = c(1, 1, 2, 2))
```

tda_manual_map

Where the manual's remaining commands live in R

Description

Working through TDA's manual section by section, some commands have no function of their own here. None of them is missing functionality; each is covered in one of three ways.

Details

Subsumed by R itself. The matrix definition commands exist because TDA has no other way to get a matrix into a session; R does. `mdef` (literal) is `matrix()`; `mdefc` (constant) is `matrix(d, m, n)`; `mdefi` (rectangular identity) is `diag(1, m, n)`; `mdeff` (from file) is `as.matrix(read.table(...))` or [tda_read_table](#); `mpr` (print a matrix expression) is printing the return value of any matrix wrapper. The expression operators of section 5.2.5 – `nd`, `ndi`, `td`, `chd`, `fd`, `mr`, `ndf`, `poisson`, `negbin`, `bivn`, `jul`, `july`, `julm`, `juld` and relatives – are `pnorm`, `qnorm`, `pt`, `pchisq`, `pf`, the Mill's ratio `dnorm(x)/(1 - pnorm(x))`, `dnorm`, `dpois/dnbinom` (TDA returns the log density), `mvtnorm::pmvnorm`, and R's date arithmetic. The engine's implementations of all of these are compared against R's on every run of the differential fuzzer, so using either side gives the same numbers; to evaluate one of TDA's operators directly, pass the expression to [tda_evalf](#).

Delivered by another wrapper. `mdefg` (adjacency matrix from the current graph) is `tda_g_edges(g, format = "full_square")`, with `-1` marking absent edges. `mbrr` is [tda_brr](#). `mple` is [tda_mple](#).

mlp's raw-tableau form is assembled internally by `tda_mlp` from the objective, constraints, and bounds.

Reached through the session they belong to. `mdefb` (a block of the data matrix) requires a `dblock` context and is used through the `commands=` argument of `tda_dblock`. `mplog`, `mppar`, `mpcov` and `mpgrad` are parameters OF estimation commands, not commands; their contents are what the fitted objects already carry as `$logLik`, `$estimates` and `$vcov` (case-specific gradients, `mpgrad`, can be requested through a command's `options=` where supported).

Chapter 5 names that are not standalone commands:

`mplog`, `mppar`, `mpcov`, `mpgrad` parameters of the estimation commands that write the log likelihood, parameters, covariance matrix or gradient into a named TDA matrix. The R wrappers return the same quantities directly (`$logLik`, `$estimates`, `$vcov`), so these are only needed inside a hand-written `tda_run` matrix pipeline, where they pass through `options=` as usual.

`mdef`, `mpr`, `mdefc`, `mdefi`, `mdeff` matrix definition and printing, which R does natively: `matrix(...)`, `print()`, `matrix(d, m, n)`, `diag(1, m, n)` and `as.matrix(read.table(fname))`; every matrix wrapper accepts plain R matrices. `mdefg` (a graph's adjacency matrix) does compute something and has its wrapper, `tda_mdefg`.

`mdefb` extracts one `dblock`-defined block of the data matrix into a matrix – in R, subset the data frame; in a `tda_run` script it works after a `dblock` command as documented.

`mbrr` the manual's name for `brr`; see `tda_brr`.

`sort (command)` see `tda_esort` and the `sort=` option of `pdata` in `tda_read_sys`.

`lag`, `suc`, `snun`, `rank`, `sort (operator)` operators of TDA's expression language, used inside `nvar` variable definitions – e.g. `L[6.0]=lag(Y,1)`, `R[6.0]=rank(Y)` – in a `tda_run` script. `snun`, `rank` and `sort` take the variable as an argument.

 tda_map |

 Maps |

Description

`tda_map` draws spatial objects in a geographical coordinate system – a projection, rather than the plain `x` and `y` of `plot()`. TDA builds that system with `psetupg` and then draws with `sdpmmap`; `graticule` adds meridians and parallels with `sdpgrat`, and `points` adds coordinates given directly with `sdpgeo`.

Usage

```
tda_map(
  s,
  view,
  region,
  projection = 10,
  width = 150,
  graticule = list(),
  points = NULL,
```

```

    symbol = 1,
    newpage = TRUE,
    ...
)

```

Arguments

<code>s</code>	a tda_spatial structure.
<code>view</code>	centre of the projection, as longitude and latitude.
<code>region</code>	half-width and half-height of the mapped area, in degrees.
<code>projection</code>	10 cylindrical equidistant, 11 cylindrical equal-area, 12 Mercator, 20 azimuthal orthographic. For projection 20 the longitude half-width is limited to 90 degrees, and the whole region rectangle must fit on the projection disc, so both halves of region need $\sin(\text{lon})^2 + \sin(\text{lat})^2 \leq 1$ at the view latitude.
<code>width</code>	width of the plot in mm.
<code>graticule</code>	a list of options for the meridians and parallels, or NULL for none. lon and lat take TDA's sequence notation, "-84(2)-76" meaning every 2 degrees from -84 to -76.
<code>points</code>	a data frame of lon and lat to mark, or NULL.
<code>symbol</code>	marker symbol for points; TDA's <code>s=</code> , which cannot be spelled that way here because <code>s</code> is the structure. 4 is an open circle, 5 a filled one, 8 a square, 15 a diamond.
<code>newpage</code>	start a new grid page.
<code>...</code>	further options for <code>sdpmap</code> .

Details

`region` is required and has no default: it is the half-width and half-height of the mapped area in degrees, measured from `view`. So `view = c(-80, 35)`, `region = c(6, 3)` maps 6 degrees of longitude and 3 of latitude either side of that point.

Value

The map, drawn; the session is returned invisibly.

See Also

Other spatial analysis: [plot.tda_spatial\(\)](#), [tda_polygons\(\)](#), [tda_read_dbf\(\)](#), [tda_read_shapefile\(\)](#), [tda_read_spatial](#), [tda_sd\(\)](#), [tda_sd_analyses](#), [tda_sd_ps3\(\)](#), [tda_spatial\(\)](#)

Examples

```

if (requireNamespace("sf", quietly = TRUE)) {
  nc <- tda_read_shapefile(system.file("shape/nc.shp", package = "sf"))
  tda_map(nc, view = c(-80, 35), region = c(6, 3),
    graticule = list(lon = "-84(2)-76", lat = "34(1)37",
      fsx = 2, fsy = 2))
}

```

tda_mat

*Matrix operations***Description**

Runs any of TDA's matrix commands on R matrices and returns R matrices. This is a generic interface rather than one function per operation: R already has `%*%`, `solve` and `eigen`, so named wrappers would add nothing, but the whole of TDA's matrix language stays reachable.

Usage

```
tda_mat(op, ..., out = "R", options = list(), dir = tempfile("tda"))
```

Arguments

<code>op</code>	the command name. TDA's matrix language has about eighty: <code>mmul</code> multiplies, <code>minvs</code> and <code>mginv</code> invert, <code>mtransp</code> transposes, <code>mev</code> and <code>mevs</code> give eigenvalues, <code>msvd</code> the singular value decomposition, <code>mchol</code> a Cholesky factor, <code>mrnk</code> the rank. Note the names: inversion is <code>minvs</code> , not <code>minv</code> , and transposition <code>mtransp</code> , not <code>mtra</code> .
<code>...</code>	input matrices, in the order the command expects.
<code>out</code>	names of the result matrices to read back.
<code>options</code>	a named list of further TDA options, passed through.
<code>dir</code>	working directory.

Value

A matrix if one result was asked for, otherwise a named list of them.

How it works

The inputs are written as TDA matrices named A, B, ... in order, the command is run, and every matrix named in `out` is read back. TDA's convention is that results come last in the argument list, so `mmul(A,B,R)` multiplies A by B into R.

```
tda_mat("mmul", A, B, out = "R")           # A
tda_mat("mev", A, out = c("ER", "EI"))    # eigenvalues, real and imaginary
```

See Also

Other smoothing: [tda_integrate\(\)](#), [tda_interp\(\)](#), [tda_isotonic\(\)](#), [tda_sma\(\)](#), [tda_smd\(\)](#), [tda_spl\(\)](#)

Examples

```
A <- matrix(1:4, 2)
B <- diag(2)
tda_mat("mmul", A, B, out = "R")
```

tda_mcatv	<i>Concatenating matrices</i>
-----------	-------------------------------

Description

tda_mcatv concatenates horizontally – TDA’s mcatv, matching cbind. tda_mcatv concatenates vertically – TDA’s mcatv, matching rbind. tda_mcatv takes the matrix direct sum: the inputs placed as block-diagonal blocks, zero elsewhere – TDA’s mcatv (all three take two or more matrices).

Usage

```
tda_mcatv(..., dir = tempfile("tda"))

tda_mcatv(..., dir = tempfile("tda"))

tda_mcatv(..., dir = tempfile("tda"))
```

Arguments

...	two or more numeric matrices.
dir	working directory.

Value

A numeric matrix.

See Also

Other matrix reshaping: [tda_mag\(\)](#), [tda_mvec\(\)](#), [tda_mrsum\(\)](#), [tda_msort\(\)](#), [tda_msrow\(\)](#), [tda_mtrim\(\)](#)

Examples

```
A <- matrix(1:4, 2); B <- matrix(5:8, 2)
tda_mcatv(A, B)    # == cbind(A, B)
tda_mcatv(A, B)    # == rbind(A, B)
tda_mcatv(A, B)    # block-diagonal direct sum
```

tda_mcel

*Build an edge list from an adjacency-style matrix***Description**

Lists every (i, j) with $A[i, j] \geq x$ as one row $(i, j, A[i, j])$ – TDA's `mcel`, scanning row by row. Refuses if no element of A reaches x (there would be nothing to return).

Usage

```
tda_mcel(A, x, dir = tempfile("tda"))
```

Arguments

`A` a square numeric matrix.
`x` the threshold; an entry is an edge when it is at least this large.
`dir` working directory.

Value

A 3-column numeric matrix: i, j , and the value $A[i, j]$, one row per edge.

See Also

Other matrix algebra: `tda_mcent()`, `tda_mch()`, `tda_mcross()`, `tda_mdiag()`, `tda_mev()`, `tda_mevs()`, `tda_midf()`, `tda_midf1()`, `tda_midf2()`, `tda_midf3()`, `tda_minvs()`, `tda_mkmet()`, `tda_mkp()`, `tda_mmul()`, `tda_mnc()`, `tda_mnrow()`, `tda_mpbl()`, `tda_mpfilt()`, `tda_mpinv()`, `tda_mpit()`, `tda_mple()`, `tda_mpz()`, `tda_mscal1()`, `tda_msqrtd()`, `tda_msvd()`, `tda_mwvec()`

Examples

```
# a valued adjacency matrix; keep the edges of value 2 or more
A <- matrix(c(0, 3, 0,
              1, 0, 5,
              0, 2, 0), 3, byrow = TRUE)
tda_mcel(A, 2) # (1,2) 3, (2,3) 5, (3,2) 2 -- (2,1) 1 is dropped
```

tda_mcent	<i>Centering, standardizing, and double-centering</i>
-----------	---

Description

tda_mcent subtracts each column's mean – TDA's mcent, matching `sweep(A, 2, colMeans(A))`. tda_mstand additionally divides each column by its *population* standard deviation (denominator n , not base R's `sd()`'s $n-1$) – TDA's mstand. tda_mdcent double-centers a symmetric matrix – TDA's mdcent, the classical (Torgerson) transformation multidimensional scaling uses to turn a matrix of *squared* distances into one whose eigendecomposition gives point coordinates.

Usage

```
tda_mcent(A, dir = tempfile("tda"))

tda_mstand(A, dir = tempfile("tda"))

tda_mdcent(D, dir = tempfile("tda"))
```

Arguments

A, D	a numeric matrix (D, for tda_mdcent, should be symmetric – typically a distance matrix).
dir	working directory.

Value

A numeric matrix.

See Also

Other matrix algebra: [tda_mcel\(\)](#), [tda_mch\(\)](#), [tda_mcross\(\)](#), [tda_mdiag\(\)](#), [tda_mev\(\)](#), [tda_mevs\(\)](#), [tda_midf\(\)](#), [tda_midf1\(\)](#), [tda_midf2\(\)](#), [tda_midf3\(\)](#), [tda_minvs\(\)](#), [tda_mkmet\(\)](#), [tda_mkp\(\)](#), [tda_mmul\(\)](#), [tda_mnc\(\)](#), [tda_mnrow\(\)](#), [tda_mpbl\(\)](#), [tda_mpfit\(\)](#), [tda_mpinv\(\)](#), [tda_mpit\(\)](#), [tda_mple\(\)](#), [tda_mpz\(\)](#), [tda_mscal1\(\)](#), [tda_msqrtd\(\)](#), [tda_msvd\(\)](#), [tda_mwvec\(\)](#)

Examples

```
A <- matrix(c(1, 2, 6, 10, 30, 20), 3, 2)
tda_mcent(A)
tda_mstand(A)
D <- matrix(c(0, 3, 4, 3, 0, 5, 4, 5, 0), 3, 3)
tda_mdcent(D)
```

tda_mch

*Check an ordering against row/column dominance***Description**

Flags each row/column index of a square matrix *A* where the current index order fails a running dominance test against its mirror image – TDA's *mch* (undocumented beyond "check row and column sum conditions" in its own header; the rule below is the loop body's, not a named textbook procedure). For each index *i*, scanning outward (*j* from *i*+1 to *n*, then separately from *i*-1 down to 1), *A*[*i*, *j*] accumulates into a running total *z* and its mirror *A*[*j*, *i*] accumulates into a running total *s*; index *i* is flagged the first time *z* falls behind *s* by more than a small tolerance. In effect: is *A*[*i*,]'s share of an outward pair always at least as large, cumulatively, as the corresponding mirrored entries in *A*[, *i*] – a diagnostic for whether the current 1:*n* ordering already respects that dominance, without finding a better ordering the way [tda_mpz](#)/ [tda_mpbl](#) do.

Usage

```
tda_mch(A, dir = tempfile("tda"))
```

Arguments

A a square numeric matrix.
dir working directory.

Value

An integer vector, length *nrow*(*A*): 1 where that index fails the test, 0 where it passes.

See Also

Other matrix algebra: [tda_mcel\(\)](#), [tda_mcent\(\)](#), [tda_mcross\(\)](#), [tda_mdiag\(\)](#), [tda_mev\(\)](#), [tda_mevs\(\)](#), [tda_midf\(\)](#), [tda_midf1\(\)](#), [tda_midf2\(\)](#), [tda_midf3\(\)](#), [tda_minvs\(\)](#), [tda_mkmet\(\)](#), [tda_mkp\(\)](#), [tda_mmul\(\)](#), [tda_mnc\(\)](#), [tda_mnrow\(\)](#), [tda_mpbl\(\)](#), [tda_mpfir\(\)](#), [tda_mpinv\(\)](#), [tda_mpit\(\)](#), [tda_mple\(\)](#), [tda_mpz\(\)](#), [tda_mscal1\(\)](#), [tda_msqrtd\(\)](#), [tda_msvd\(\)](#), [tda_mwvec\(\)](#)

Examples

```
# A[i, j] = 5 means i strongly precedes j. Ordered 1 < 2 < 3 the
# rows dominate their columns and every index passes (0); the same
# matrix with the order reversed fails everywhere (1).
A <- matrix(c(0, 5, 5,
              1, 0, 5,
              1, 1, 0), 3, byrow = TRUE)

tda_mch(A)
tda_mch(A[3:1, 3:1])
```

tda_mcross

*Cross-product and an arithmetic sequence as a column***Description**

tda_mcross is TDA's mcross, matching base R's crossprod(A) (t(A) %*% A). tda_mnum builds the length-n arithmetic sequence starting at x with step d – TDA's mnum, matching seq(x, by = d, length.out = n).

Usage

```
tda_mcross(A, dir = tempfile("tda"))

tda_mnum(x, d, n, dir = tempfile("tda"))
```

Arguments

A	a numeric matrix.
dir	working directory.
x	the starting value, for tda_mnum.
d	the step, for tda_mnum.
n	the length, for tda_mnum.

Value

tda_mcross returns a matrix; tda_mnum returns a plain numeric vector.

See Also

Other matrix algebra: [tda_mcel\(\)](#), [tda_mcent\(\)](#), [tda_mch\(\)](#), [tda_mdiag\(\)](#), [tda_mev\(\)](#), [tda_mevs\(\)](#), [tda_midf\(\)](#), [tda_midf1\(\)](#), [tda_midf2\(\)](#), [tda_midf3\(\)](#), [tda_minvs\(\)](#), [tda_mkmet\(\)](#), [tda_mkp\(\)](#), [tda_mmul\(\)](#), [tda_mnc\(\)](#), [tda_mnrow\(\)](#), [tda_mpb1\(\)](#), [tda_mpf1t\(\)](#), [tda_mpinv\(\)](#), [tda_mpit\(\)](#), [tda_mple\(\)](#), [tda_mpz\(\)](#), [tda_mscal1\(\)](#), [tda_msqrtd\(\)](#), [tda_msvd\(\)](#), [tda_mwvec\(\)](#)

Examples

```
tda_mcross(matrix(1:6, 3, 2)) # == crossprod(matrix(1:6, 3, 2))
tda_mnum(5, 2, 4)             # == seq(5, by = 2, length.out = 4)
```

tda_mcvec

*Vectorizing a matrix and its inverse***Description**

tda_mcvec stacks the columns of A into a single vector – TDA’s mcvec, the classic matrix-algebra `vec(A)` operator, matching base R’s `as.vector(A)` (R matrices are column-major already). `tda_mrvec` stacks the rows instead – TDA’s mrvec, matching `as.vector(t(A))`. `tda_mivec` is the inverse of `tda_mcvec`: given a vector and a row count n, it rebuilds an n-row matrix by filling column by column – TDA’s mivec, matching base R’s `matrix(x, nrow = n)`.

Usage

```
tda_mcvec(A, dir = tempfile("tda"))

tda_mrvec(A, dir = tempfile("tda"))

tda_mivec(x, n, dir = tempfile("tda"))
```

Arguments

A	a numeric matrix.
dir	working directory.
x	a numeric vector, whose length must divide evenly by n.
n	the number of rows for <code>tda_mivec</code> ’s result.

Value

`tda_mcvec`/`tda_mrvec` return a plain numeric vector; `tda_mivec` returns a matrix.

See Also

Other matrix reshaping: [tda_mag\(\)](#), [tda_mcat\(\)](#), [tda_mrsum\(\)](#), [tda_msort\(\)](#), [tda_msrow\(\)](#), [tda_mtrim\(\)](#)

Examples

```
A <- matrix(1:6, 2, 3)
tda_mcvec(A)           # == as.vector(A)
tda_mrvec(A)           # == as.vector(t(A))
tda_mivec(1:6, 2)      # == matrix(1:6, nrow = 2)
```

tda_mdefg	<i>Adjacency matrix of a graph</i>
-----------	------------------------------------

Description

TDA's mdefg: builds the adjacency matrix of the current graph, with missing substituted for absent edges (TDA's sc=, default -1 there; 0 here, which gives the usual weighted adjacency matrix).

Usage

```
tda_mdefg(edges, missing = 0, directed = FALSE, valued = TRUE, ...)
```

Arguments

edges	edge list (data frame with from, to and optionally a value column).
missing	value written where no edge exists.
directed, valued	graph type flags, as in tda_g .
...	passed to tda_run .

Value

the numeric adjacency matrix.

Examples

```
e <- data.frame(from = c(1, 1, 2), to = c(2, 3, 3),
                 value = c(5, 2, 7))
tda_mdefg(e)
```

tda_mdiag	<i>Diagonal construction and extraction</i>
-----------	---

Description

tda_mdiag builds an n x n diagonal matrix from a length-n vector – TDA's mdiag, matching base R's diag(x) when x has length > 1. tda_mdiagd extracts the diagonal of a matrix as a plain vector, using min(nrow(A), ncol(A)) entries for a non-square A – TDA's mdiagd, matching base R's diag(A).

Usage

```
tda_mdiag(x, dir = tempfile("tda"))

tda_mdiagd(A, dir = tempfile("tda"))
```

Arguments

`x` a numeric vector.
`dir` working directory.
`A` a numeric matrix.

Value

`tda_mdiag` returns a matrix; `tda_mdiagd` returns a plain numeric vector.

See Also

Other matrix algebra: `tda_mcel()`, `tda_mcent()`, `tda_mch()`, `tda_mcross()`, `tda_mev()`, `tda_mevs()`, `tda_midf()`, `tda_midf1()`, `tda_midf2()`, `tda_midf3()`, `tda_minvs()`, `tda_mkmet()`, `tda_mkp()`, `tda_mmul()`, `tda_mnc()`, `tda_mnrow()`, `tda_mpb1()`, `tda_mpf1t()`, `tda_mpinv()`, `tda_mpit()`, `tda_mple()`, `tda_mpz()`, `tda_mscal1()`, `tda_msqrtd()`, `tda_msvd()`, `tda_mwvec()`

Examples

```
tda_mdiag(c(5, 7, 9))
tda_mdiagd(matrix(1:6, 2, 3))
```

TDA_MDS

Multidimensional scaling

Description

Represent a dissimilarity matrix as points in a low-dimensional space, so that the distances between the points approximate the dissimilarities.

Usage

```
TDA_MDS

tda_mds(
  d,
  method = "classical",
  ndim = 2,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

<code>d</code>	a dist, or a square dissimilarity matrix.
<code>method</code>	one of the names in TDA_MDS, matched partially.
<code>ndim</code>	number of dimensions to return.
<code>options</code>	a named list of further TDA options, passed through.
<code>dir</code>	working directory.
<code>...</code>	passed to <code>tda_run</code> .

Details

`method = "classical"` (TDA's `mdsc`) is the eigenvalue decomposition also known as principal coordinates analysis – the same method as `cmdscale`. TDA computes the full solution, one dimension per positive eigenvalue, and never reads a dimension count of its own; because the classical solution is nested, the first `ndim` columns of the full solution *are* the `ndim`-dimensional solution, so this wrapper simply keeps those. The eigenvalues, with the share of the total each accounts for, come back in `eigenvalues` – the usual guide to how many dimensions the data support.

The other three methods fit the configuration iteratively. `"metric"` (`mdsm`) minimizes the raw stress, the sum of squared differences between dissimilarities and fitted distances, with TDA's general minimizer; it fits two dimensions only, and errors for any other `ndim`. `"nonmetric"` (`mdsn`) minimizes Kruskal's stress-1, preserving only the rank order of the dissimilarities, by Kruskal's (1964) gradient method with his step length adaptation; it reaches the same or lower stress than `isoMDS` on the usual test data, but its default `mxit = 100` iterations is often not enough – pass `options = list(mxit = 300)`, and `ns = random` restarts to guard against local minima. `"minmax"` (`mdsx`) fits a minmax criterion; TDA labels that command experimental and its output is left as it comes. Check stress before trusting any iterative configuration.

Value

An object carrying points, the coordinates (one row per object, one column per dimension), and, for `"classical"`, eigenvalues (a data frame of each eigenvalue and its percentage of the total); for the iterative methods, stress, the criterion value at the returned configuration.

See Also

Other clustering: `TDA_CLUSTER`, `tda_conjoint()`, `tda_cutree()`, `tda_dma()`, `tda_pdatd()`

Examples

```
set.seed(34)
pts <- rbind(matrix(rnorm(10, 0), 5, 2), matrix(rnorm(10, 6), 5, 2))
m <- tda_mds(dist(pts), method = "classical", ndim = 2)
m$points      # recovers the two clusters as two groups of coordinates
m$eigenvalues # two dominant dimensions, as built
```

tda_mdsn

*Nonmetric multidimensional scaling***Description**

TDA's mdsn1 (Kruskal) on a distance matrix.

Usage

```
tda_mdsn(
  d,
  ties = c("ignore", "primary", "secondary"),
  max_iterations = 100,
  restarts = 5,
  ...
)
```

Arguments

<code>d</code>	symmetric distance matrix.
<code>ties</code>	tie handling. Only "ignore" is implemented in mdsn1 (its primary and secondary projections are commented out in the C, and the option would silently do nothing); the other two values error and point to tda_mds , whose mdsn has them.
<code>max_iterations</code>	iteration cap.
<code>restarts</code>	random starting configurations, TDA's ns=; the one with the lowest stress is returned. Nonmetric MDS on few points has degenerate local minima (clusters of near-coincident points with small stress), and restarts are the guard against them.
<code>...</code>	passed to tda_run .

Value

with direct exports enabled, a list with stress, configuration (n x 2) and output; otherwise the printed output lines.

Examples

```
d <- as.matrix(dist(cbind(c(0, 0, 3, 3), c(0, 2, 0, 2))))
tda_mdsn(d)
```

tda_mdsr	<i>Axes in an MDS configuration</i>
----------	-------------------------------------

Description

TDA's mdsr: regresses external variables onto a two-dimensional configuration to draw interpretable axes.

Usage

```
tda_mdsr(config, x, ...)
```

Arguments

config	two-column matrix, the configuration.
x	matrix of external variables (same row count).
...	passed to tda_run .

Value

the printed output.

Examples

```
cfg <- cbind(c(0, 1, 2, 3), c(0, 1, 0, 1))
r <- tda_mdsr(cfg, x = cbind(v = c(1, 2, 3, 4)))
cat(head(tda_payload(r), 12), sep = "\n")
```

tda_mev	<i>Eigen decomposition of a general (possibly non-symmetric) matrix</i>
---------	---

Description

Eigenvalues and eigenvectors of a square matrix that need not be symmetric, so eigenvalues and eigenvectors may be complex – TDA's mev. `t_mat.c`'s dispatch comment understates this command's arity (`mev(A, ER, EI, EV)`, four arguments); what `m_mev` actually parses is five (`mev(A, ER, EI, EVR, EVI)`), splitting the eigenvector matrix into separate real and imaginary parts, as `eigen1`'s header lists them. Column `i` of the result, `EVR[, i] + 1i * EVI[, i]`, is the right eigenvector for eigenvalue `ER[i] + 1i * EI[i]`, i.e. $A \cdot v = \lambda \cdot v$ – the same convention and ordering as base R's `eigen(A)`, confirmed on two matrices, one with real eigenvalues and one with an asymmetric complex pair (not simply plus/minus of each other, which the first, simpler test matrix happened to have and which turned out to mask a construction mistake in an earlier verification attempt – recorded here because it is exactly the kind of false confirmation a too-simple or too-symmetric test case can produce, the same lesson [tda_mevs](#) needed).

Usage

```
tda_mev(A, dir = tempfile("tda"))
```

Arguments

- A a square numeric matrix.
- dir working directory.

Details

TDA's `mev(A, ER, EI, EVR, EVI)` requires all four output operands (none are optional); the wrapper supplies them and folds the real and imaginary parts into complex `$values` and `$vectors`.

Value

A list: `values`, a complex vector of eigenvalues; `vectors`, a complex matrix whose columns are the corresponding right eigenvectors.

See Also

Other matrix algebra: `tda_mcel()`, `tda_mcent()`, `tda_mch()`, `tda_mcross()`, `tda_mdiag()`, `tda_mevs()`, `tda_midf()`, `tda_midf1()`, `tda_midf2()`, `tda_midf3()`, `tda_minvs()`, `tda_mkmet()`, `tda_mkp()`, `tda_mmul()`, `tda_mnc()`, `tda_mnrow()`, `tda_mpb1()`, `tda_mpf1t()`, `tda_mpinv()`, `tda_mpit()`, `tda_mple()`, `tda_mpz()`, `tda_mscal1()`, `tda_msqrtd()`, `tda_msvd()`, `tda_mwvec()`

Examples

```
tda_mev(matrix(c(0, -1, 1, 0), 2, 2)) # eigenvalues +/-i
```

tda_mevs	<i>Eigen decomposition of a symmetric matrix</i>
----------	--

Description

Eigenvalues and eigenvectors of a symmetric matrix – TDA's `mevs`. Eigenvalues come back in *descending* order, same as base R's `eigen(A, symmetric = TRUE)`. The eigenvectors come back as the *columns* of `vectors`, same convention as `eigen()$vectors` (column `i` satisfies `A %*% vectors[, i] == values[i] * vectors[, i]`); an initial 2x2 test matrix was checked first and satisfied this equally well as rows, which turned out to be a coincidence of that specific matrix's eigenvector matrix happening to be symmetric, not a real property of TDA's convention – worth recording since it is exactly the kind of false confirmation a too-simple test case can produce. As with any eigenvector, the sign of each one is arbitrary; TDA's and R's choices need not agree.

Usage

```
tda_mevs(A, dir = tempfile("tda"))
```

Arguments

A a symmetric numeric matrix (only the lower triangle is read).
 dir working directory.

Value

A list: values, the eigenvalues in descending order; vectors, a matrix whose *columns* are the corresponding unit-norm eigenvectors.

See Also

Other matrix algebra: `tda_mcel()`, `tda_mcent()`, `tda_mch()`, `tda_mcross()`, `tda_mdiag()`, `tda_mev()`, `tda_midf()`, `tda_midf1()`, `tda_midf2()`, `tda_midf3()`, `tda_minvs()`, `tda_mkmet()`, `tda_mkp()`, `tda_mmul()`, `tda_mnc()`, `tda_mnrow()`, `tda_mpbl()`, `tda_mpfite()`, `tda_mpinv()`, `tda_mpit()`, `tda_mple()`, `tda_mpz()`, `tda_mscal1()`, `tda_msqrtd()`, `tda_msvd()`, `tda_mwvec()`

Examples

```
A <- matrix(c(4, 1, 0,
              1, 3, 1,
              0, 1, 2), 3)
e <- tda_mevs(A)
e$values                    # descending, as eigen(A)$values
A %*% e$vectors[, 1] - e$values[1] * e$vectors[, 1]    # zero: an eigenpair
```

tda_midf

*Distribution function bounds and estimate for interval-censored data***Description**

Given a sample of intervals [lower_i, upper_i] (each interval containing an unobserved exact value), estimates the distribution function of that unobserved value – TDA’s midf (undocumented beyond “calculates distribution function DM, and lower/upper bounds” in its own header, which also understates the command’s arity: it lists three outputs, midf(XL,XU,DL,DU,DM), but m_midf actually returns four, midf(XL,XU,PT,DL,DU,DM) – the breakpoints themselves, PT, are a result too). Evaluated at every distinct interval endpoint (the “induced partition” breakpoints): lower_bound counts only intervals entirely below that point (a sure lower bound on the true CDF); upper_bound counts every interval that could be below it (a sure upper bound); cdf is a point estimate in between, assuming each interval’s unobserved value is uniformly distributed within it. Every interval must have positive width (lower < upper, swapped automatically if given in the other order). Confirmed cell-for-cell by hand on a 3-interval example.

Usage

```
tda_midf(lower, upper, dir = tempfile("tda"))
```

Arguments

lower, upper numeric vectors, the same length, with lower[i] != upper[i] for every i.
 dir working directory.

Value

A list: breakpoints, the sorted distinct interval endpoints; lower_bound, upper_bound, cdf, each the same length as breakpoints.

See Also

Other matrix algebra: [tda_mcel\(\)](#), [tda_mcent\(\)](#), [tda_mch\(\)](#), [tda_mcross\(\)](#), [tda_mdiag\(\)](#), [tda_mev\(\)](#), [tda_mevs\(\)](#), [tda_midf1\(\)](#), [tda_midf2\(\)](#), [tda_midf3\(\)](#), [tda_minvs\(\)](#), [tda_mkmet\(\)](#), [tda_mkp\(\)](#), [tda_mmul\(\)](#), [tda_mnc\(\)](#), [tda_mnrow\(\)](#), [tda_mpbl\(\)](#), [tda_mpfit\(\)](#), [tda_mpinv\(\)](#), [tda_mpit\(\)](#), [tda_mple\(\)](#), [tda_mpz\(\)](#), [tda_mscal1\(\)](#), [tda_msqrtd\(\)](#), [tda_msvd\(\)](#), [tda_mwvec\(\)](#)

Examples

```
# five observations known only to intervals: [0,2], [1,3], [2,4], [3,5], [1,2]
lo <- c(0, 1, 2, 3, 1)
up <- c(2, 3, 4, 5, 2)
# at each endpoint: the share of intervals surely below it (lower
# bound), possibly below it (upper bound), and the estimate between
tda_midf(lo, up)
```

tda_midf1

Uniform-model CDF for interval-censored data, at each observation

Description

The same uniform-within-interval distribution function estimate as [tda_midf](#)'s cdf, but evaluated at each observation's lower[i] rather than at every distinct breakpoint – TDA's midf1.

Usage

```
tda_midf1(lower, upper, dir = tempfile("tda"))
```

Arguments

lower, upper numeric vectors, the same length, with lower[i] != upper[i] for every i.
 dir working directory.

Value

A numeric vector, the same length as lower, in the original (unsorted) order.

See Also

Other matrix algebra: `tda_mcel()`, `tda_mcent()`, `tda_mch()`, `tda_mcross()`, `tda_mdiag()`, `tda_mev()`, `tda_mevs()`, `tda_midf()`, `tda_midf2()`, `tda_midf3()`, `tda_minvs()`, `tda_mkmet()`, `tda_mkp()`, `tda_mmul()`, `tda_mnc()`, `tda_mnrow()`, `tda_mpbl()`, `tda_mpfite()`, `tda_mpinv()`, `tda_mpit()`, `tda_mple()`, `tda_mpz()`, `tda_mscal1()`, `tda_msqrtd()`, `tda_msvd()`, `tda_mwvec()`

Examples

```
lo <- c(0, 1, 2, 3, 1)
up <- c(2, 3, 4, 5, 2)
# the estimated distribution function at each interval's own position
tda_midf1(lo, up)
```

tda_midf2

*Restricted mean for interval-censored data, at each observation***Description**

For each observation, the conditional mean of the unobserved value given that it exceeds that observation's $\text{lower}[i] - E[X \mid X > \text{lower}_i]$ – estimated by integrating a piecewise-uniform density built from `tda_midf`'s CDF estimate over the induced partition – TDA's `midf2`: it builds the CDF at every breakpoint, then integrates the resulting piecewise-constant density from each observation's value onward, normalizing by the survival probability there.

Usage

```
tda_midf2(lower, upper, dir = tempfile("tda"))
```

Arguments

`lower`, `upper` numeric vectors, the same length, with `lower[i] != upper[i]` for every `i`.
`dir` working directory.

Value

A numeric vector, the same length as `lower`, in the original (unsorted) order.

See Also

Other matrix algebra: `tda_mcel()`, `tda_mcent()`, `tda_mch()`, `tda_mcross()`, `tda_mdiag()`, `tda_mev()`, `tda_mevs()`, `tda_midf()`, `tda_midf1()`, `tda_midf3()`, `tda_minvs()`, `tda_mkmet()`, `tda_mkp()`, `tda_mmul()`, `tda_mnc()`, `tda_mnrow()`, `tda_mpbl()`, `tda_mpfite()`, `tda_mpinv()`, `tda_mpit()`, `tda_mple()`, `tda_mpz()`, `tda_mscal1()`, `tda_msqrtd()`, `tda_msvd()`, `tda_mwvec()`

Examples

```
lo <- c(0, 1, 2, 3, 1)
up <- c(2, 3, 4, 5, 2)
# the expected value of each observation given that it lies in its
# interval, under the estimated distribution
tda_midf2(lo, up)
```

tda_midf3

*Average nearby endpoints for interval-censored data***Description**

For each interval i , averages the lower endpoints of every interval (itself included) whose lower endpoint falls within $[lower[i], upper[i]]$, and separately averages the upper endpoints falling in that same range – TDA's midf3. A local smoothing of nearby endpoints, not a distribution estimate like [tda_midf](#)/[tda_midf1](#)/[tda_midf2](#).

Usage

```
tda_midf3(lower, upper, dir = tempfile("tda"))
```

Arguments

`lower, upper` numeric vectors, the same length.
`dir` working directory.

Value

A list, both the same length as `lower`, in the original order: `lower_avg`, the average nearby lower endpoint; `upper_avg`, the average nearby upper endpoint.

See Also

Other matrix algebra: [tda_mcel\(\)](#), [tda_mcent\(\)](#), [tda_mch\(\)](#), [tda_mcross\(\)](#), [tda_mdiag\(\)](#), [tda_mev\(\)](#), [tda_mevs\(\)](#), [tda_midf\(\)](#), [tda_midf1\(\)](#), [tda_midf2\(\)](#), [tda_minvs\(\)](#), [tda_mkmet\(\)](#), [tda_mkp\(\)](#), [tda_mmul\(\)](#), [tda_mnc\(\)](#), [tda_mnrow\(\)](#), [tda_mpbl\(\)](#), [tda_mpfite\(\)](#), [tda_mpinv\(\)](#), [tda_mpit\(\)](#), [tda_mple\(\)](#), [tda_mpz\(\)](#), [tda_mscal1\(\)](#), [tda_msqrtd\(\)](#), [tda_msvd\(\)](#), [tda_mwvec\(\)](#)

Examples

```
lo <- c(0, 1, 2, 3, 1)
up <- c(2, 3, 4, 5, 2)
# for each interval, the mean lower and upper endpoint of the
# intervals overlapping it
tda_midf3(lo, up)
```

tda_minvs

*Matrix inversion***Description**

Three different things, despite the similar names – read the one you need. `tda_minvs` is the actual matrix inverse of a symmetric positive-definite matrix (TDA's `minvs`, via a Cholesky-based algorithm; refuses if not positive definite). `tda_minvd` inverts only the diagonal elements of `A`, element by element – *not* a matrix inverse at all, and off-diagonal entries are ignored entirely (TDA's `minvd`; a zero diagonal entry is silently left as 0 rather than inverted). `tda_mginv` is the Moore-Penrose pseudoinverse, requiring at least as many rows as columns and full column rank (TDA's `mginv`; prints the pseudorank as a side effect, discarded here – read it from `attr(x, "rank")$output` if needed).

Usage

```
tda_minvs(A, dir = tempfile("tda"))
```

```
tda_minvd(A, dir = tempfile("tda"))
```

```
tda_mginv(A, dir = tempfile("tda"))
```

Arguments

`A` a numeric matrix: square for `tda_minvs`; any shape for `tda_minvd`; at least as many rows as columns for `tda_mginv`.

`dir` working directory.

Value

A numeric matrix.

See Also

Other matrix algebra: `tda_mcel()`, `tda_mcent()`, `tda_mch()`, `tda_mcross()`, `tda_mdiag()`, `tda_mev()`, `tda_mevs()`, `tda_midf()`, `tda_midf1()`, `tda_midf2()`, `tda_midf3()`, `tda_mkmet()`, `tda_mkp()`, `tda_mmul()`, `tda_mnc()`, `tda_mnrow()`, `tda_mpbl()`, `tda_mpfit()`, `tda_mpinv()`, `tda_mpit()`, `tda_mple()`, `tda_mpz()`, `tda_mscal1()`, `tda_msqrtd()`, `tda_msvd()`, `tda_mwvec()`

Examples

```
A <- matrix(c(4, 2, 2, 3), 2)
tda_minvs(A) # == solve(A) for symmetric positive-definite A
tda_minvd(matrix(c(2, 1, 4, 5, 3, 8, 9, 7, 1), 3)) # diag(1/diag(.))
tda_mginv(matrix(c(1, 1, 1, 1, 2, 3), 3))
```

tda_mkmet

*Kemeny distance between rankings***Description**

Pairwise Kemeny (Kendall-tau-with-ties) distance between every pair of rows of A, each row a ranking (or score) over the same set of items – TDA's mkmet. For two rankings, each pair of items contributes 0 if both rankings order (or tie) the pair the same way, 1 if one ranking ties the pair and the other doesn't, or 2 if the rankings strictly disagree on the pair's order.

Usage

```
tda_mkmet(A, dir = tempfile("tda"))
```

Arguments

A a numeric matrix, one ranking per row.
 dir working directory.

Value

A symmetric numeric matrix with zero diagonal, the Kemeny distance between every pair of rows.

See Also

Other matrix algebra: [tda_mcel\(\)](#), [tda_mcent\(\)](#), [tda_mch\(\)](#), [tda_mcross\(\)](#), [tda_mdiag\(\)](#), [tda_mev\(\)](#), [tda_mevs\(\)](#), [tda_midf\(\)](#), [tda_midf1\(\)](#), [tda_midf2\(\)](#), [tda_midf3\(\)](#), [tda_minvs\(\)](#), [tda_mkp\(\)](#), [tda_mmul\(\)](#), [tda_mnc\(\)](#), [tda_mnrow\(\)](#), [tda_mpbl\(\)](#), [tda_mpfit\(\)](#), [tda_mpinv\(\)](#), [tda_mpit\(\)](#), [tda_mple\(\)](#), [tda_mpz\(\)](#), [tda_mscal1\(\)](#), [tda_msqrtd\(\)](#), [tda_msvd\(\)](#), [tda_mwvec\(\)](#)

Examples

```
# four rankings of the same four items: b swaps the last two of a,
# c reverses a, d ties items in pairs
R <- rbind(a = c(1, 2, 3, 4),
           b = c(1, 2, 4, 3),
           c = c(4, 3, 2, 1),
           d = c(1, 1, 2, 2))
tda_mkmet(R) # a-b: 2 (one pair reversed); a-c: 12 (all six pairs)
```

tda_mkp	<i>Kronecker product</i>
---------	--------------------------

Description

TDA's mkp, matching base R's kronecker(A, B) exactly.

Usage

```
tda_mkp(A, B, dir = tempfile("tda"))
```

Arguments

A, B	numeric matrices.
dir	working directory.

Value

A numeric matrix, nrow(A)*nrow(B) by ncol(A)*ncol(B).

See Also

Other matrix algebra: [tda_mcel\(\)](#), [tda_mcent\(\)](#), [tda_mch\(\)](#), [tda_mcross\(\)](#), [tda_mdiag\(\)](#), [tda_mev\(\)](#), [tda_mevs\(\)](#), [tda_midf\(\)](#), [tda_midf1\(\)](#), [tda_midf2\(\)](#), [tda_midf3\(\)](#), [tda_minvs\(\)](#), [tda_mkmet\(\)](#), [tda_mmul\(\)](#), [tda_mnc\(\)](#), [tda_mnrow\(\)](#), [tda_mpbl\(\)](#), [tda_mpf1t\(\)](#), [tda_mpinv\(\)](#), [tda_mpit\(\)](#), [tda_mple\(\)](#), [tda_mpz\(\)](#), [tda_mscal1\(\)](#), [tda_msqrtd\(\)](#), [tda_msvd\(\)](#), [tda_mwvec\(\)](#)

Examples

```
tda_mkp(matrix(1:4, 2), matrix(5:8, 2))
```

tda_mlp	<i>Linear programming</i>
---------	---------------------------

Description

Maximize (or minimize) a linear objective subject to linear inequality and/or equality constraints – TDA's mlp and mlp1 (Salazar & Sen's MINIT, CACM algorithm 333). tda_mlp takes only inequality constraints, matching mlp(T, X, Y); tda_mlp1 adds equality constraints, matching mlp1(T, p, X, Y) – the two are the same underlying call with p fixed at 0 for tda_mlp. The problem solved, after negating the objective internally for direction = "min", is

$$\max c'x \quad \text{s.t. } x \geq 0, Ax \leq b, A_e x = b_e$$

with the dual $\min b'y$ s.t. $y \geq 0, A'y \geq c$ (this is all stated directly in lp1's header in t_lp.c; nothing here was guessed). Equality-constraint rows report a dual value of 0 – TDA's convention, not something this wrapper computes. Verified by hand for a plain and an equality instance, and against brute-force vertex enumeration in test-r-comparisons.R.

Usage

```
tda_mlp(
  objective,
  constraints,
  bounds,
  direction = c("max", "min"),
  dir = tempfile("tda"),
  ...
)

tda_mlp1(
  objective,
  constraints = NULL,
  bounds = NULL,
  equalities = NULL,
  equalities_bounds = NULL,
  direction = c("max", "min"),
  dir = tempfile("tda"),
  ...
)
```

Arguments

objective	coefficients c , one per variable (at least one).
constraints	the inequality constraint matrix A ($Ax \leq \text{bounds}$), one row per constraint, one column per variable. For <code>tda_mlp1</code> , may be omitted if equalities is given instead.
bounds	the right-hand sides b for constraints.
direction	"max" (default) or "min".
dir	working directory.
...	passed to <code>tda_run</code> .
equalities	an equality constraint matrix A_e ($A_e x = \text{equalities_bounds}$), same number of columns as constraints. <code>tda_mlp1</code> only.
equalities_bounds	the right-hand sides b_e for equalities. <code>tda_mlp1</code> only.

Details

`tda_mlp` also accepts TDA's single-tableau form, `mlp(T, X, Y)`: the first row is the objective (c , $\emptyset$) and the remaining rows are the constraints $[A, b]$, exactly the layout the solver documents. `tda_mlp(T)` decomposes it accordingly.

The manual's Box example writes `mlp(T, X, Y, 1)` with a fourth operand. That form does not exist in this source tree: the shipped 6.4q parser (checked against a pristine copy of the upstream source) accepts exactly `mlp(T, X, Y)` and `mlp1(T, p, X, Y)`, so the box was produced by a different release. On the box's problem the three-operand form gives the same value (1) and the same dual vector (1, 0); only the primal solution differs – (1, 0) here against the manual's (1, 1) – and both are

vertices of the same optimal face, since maximising x_1 under $x_1 \leq 1$, $x_2 \leq 1$ leaves x_2 free at the optimum. Nothing disagrees beyond the tie-break.

Value

A list: value, the objective at the optimum; x, the primal solution; y, the dual solution (one entry per constraint row, inequalities then equalities, equality rows always 0).

See Also

Other optimization: [tda_boolean_min\(\)](#), [tda_mlp1\(\)](#), [tda_mls\(\)](#), [tda_mqap\(\)](#), [tda_mqp\(\)](#)

Examples

```
# maximize 3x1 + 2x2 s.t. x1 + x2 <= 4, x1 + 3x2 <= 6
tda_mlp(c(3, 2), rbind(c(1, 1), c(1, 3)), c(4, 6))
# same, plus x1 + x2 = 3 exactly
tda_mlp1(c(3, 2), rbind(c(1, 3)), c(6), equalities = rbind(c(1, 1)),
         equalities_bounds = 3)
```

tda_mlp1

0-1 linear programming

Description

Maximize a linear objective over 0-1 variables under linear constraints – TDA’s mlp1 (CACM algorithm 449). The problem solved is

$$\max c'x + k \quad \text{s.t. } Ax \geq b, x \in \{0, 1\}^n$$

with integer coefficients (negative ones are fine). The command’s source states no objective; this is what it computes, established against exhaustive search – see [examples/coverage/mlpiops.cf](#).

Usage

```
tda_mlp1(
  objective,
  constraints,
  bounds,
  constant = 0,
  max_solutions = 100,
  dir = tempfile("tda"),
  ...
)
```

Arguments

objective	integer coefficients <i>c</i> , one per variable (at least two variables).
constraints	the constraint matrix <i>A</i> , one row per constraint (at least one), <code>length(objective)</code> columns.
bounds	the right-hand sides <i>b</i> , one per constraint row.
constant	<i>k</i> , added to the reported value (default 0).
max_solutions	how many optimal solutions to return at most.
dir	working directory.
...	passed to <code>tda_run</code> .

Value

A list: `value`, the maximum; `solutions`, a matrix with one optimal 0-1 vector per row.

See Also

Other optimization: `tda_boolean_min()`, `tda_mlp()`, `tda_mls()`, `tda_mqap()`, `tda_mqp()`

Examples

```
# maximize 2 x1 - 3 x2 + 4 x3 + x4 s.t. sum(x) >= 2,
# -x1 + 2 x2 + x4 >= 1
tda_mlp(c(2, -3, 4, 1),
        rbind(c(1, 1, 1, 1), c(-1, 2, 0, 1)),
        bounds = c(2, 1))
```

 tda_mlrc_design | *Design matrix for MLRC models* |**Description**

The within-group pairwise design matrix TDA's `mlides` builds for MLRC models: for objects with attribute rows *z* and a grouping in consecutive runs, the row for the ordered pair (*kk*, *k*) holds the products `z[k, i1] * z[kk, i2]` (*i2*-major) plus an indicator for `k == kk` when both objects are in the same group, and zeros otherwise. Semantics established against the source and verified cell-for-cell; see `examples/coverage/mlidesops.cf`.

Usage

```
tda_mlrc_design(z, groups, dir = tempfile("tda"), ...)
```

Arguments

<i>z</i>	an <i>n</i> x <i>q</i> matrix, one row per object.
<i>groups</i>	length- <i>n</i> vector; consecutive equal values form groups.
<i>dir</i>	working directory.
...	passed to <code>tda_run</code> .

Value

The $n^2 \times (q^2 + 1)$ design matrix.

See Also

Other regression: [TDA_FAMILIES](#), [TDA_QRMODELS](#), [tda_freg\(\)](#), [tda_gdf\(\)](#), [tda_l1reg\(\)](#), [tda_lsreg\(\)](#), [tda_mreg\(\)](#), [tda_nlreg\(\)](#), [tda_npreg\(\)](#), [tda_zreg\(\)](#)

Examples

```
z <- rbind(c(1, 2), c(3, 4), c(5, 6))
tda_mlrc_design(z, groups = c(1, 1, 2))
```

tda_mls	<i>Constrained least squares</i>
---------	----------------------------------

Description

Least squares with optional linear equality constraints, linear inequality constraints, and non-negativity – TDA’s `mls`, `mlse`, `mlsi`, `mlsei`, `mlsei1`, and `mnls`, each as its thin wrapper sharing one implementation. The problem solved is

$$\min \|Ax - b\|_2 \quad \text{s.t. } A_e x = b_e, A_i x \geq b_i, (x \geq 0)$$

These are TDA’s documented semantics (`tda.hlp`, entries `##mls`/`##mlse`/`##mlsi`/`##mlsei`/`##mnls`): an earlier working note had assumed `mlse`/`mlsi` took their constraint counts, which the source does not support – `tda_mlse` and `tda_mls` turn out to make the identical call (confirmed empirically: an overdetermined, inconsistent instance returns the plain least-squares fit rather than an error), and `tda_mlsi` treats its entire input as inequality rows. `tda_mlsei` and `tda_mlsei1` solve the same general equality+least-squares+inequality problem via two different TDA algorithms (Lawson-Hanson vs. an undocumented active-set QP, confirmed to agree to solver tolerance on two independent instances, not merely assumed from the similar name). `tda_mnls` is the only one of the six with non-negativity, applied to every variable (TDA’s `k` split point fixed at 0) and cannot be combined with inequalities (`mnls` has no general inequality block of its own).

Usage

```
tda_mls(A, b, dir = tempfile("tda"), ...)

tda_mlse(A, b, dir = tempfile("tda"), ...)

tda_mlsi(inequalities, inequalities_bounds, dir = tempfile("tda"), ...)

tda_mlsei(
  A = NULL,
  b = NULL,
  equalities = NULL,
```

```

    equalities_bounds = NULL,
    inequalities = NULL,
    inequalities_bounds = NULL,
    dir = tempfile("tda"),
    ...
)

tda_mlsei1(
    A = NULL,
    b = NULL,
    equalities = NULL,
    equalities_bounds = NULL,
    inequalities = NULL,
    inequalities_bounds = NULL,
    dir = tempfile("tda"),
    ...
)

tda_mnls(
    A,
    b,
    equalities = NULL,
    equalities_bounds = NULL,
    dir = tempfile("tda"),
    ...
)

```

Arguments

<code>A</code>	the least-squares data matrix ($m \times n$). Required for <code>tda_mls</code> / <code>tda_mlse</code> / <code>tda_mnls</code> ; optional for <code>tda_mlsei</code> / <code>tda_mlsei1</code> (NULL for a pure feasibility problem, together with equalities and/or inequalities); not used by <code>tda_mlsi</code> .
<code>b</code>	the target vector for $Ax \sim b$, length m .
<code>dir</code>	working directory.
<code>...</code>	passed to tda_run .
<code>inequalities</code>	an inequality constraint matrix $A_i x \geq inequalities_bounds$. <code>tda_mlsi</code> (required, its only input) and <code>tda_mlsei</code> / <code>tda_mlsei1</code> (optional).
<code>inequalities_bounds</code>	the right-hand sides for inequalities.
<code>equalities</code>	an equality constraint matrix $A_e x = equalities_bounds$. <code>tda_mlsei</code> / <code>tda_mlsei1</code> / <code>tda_mnls</code> only.
<code>equalities_bounds</code>	the right-hand sides for equalities.

Details

`tda_mls`, `tda_mlse` and `tda_mlsi` also accept the manual's single-matrix form, `mls(S, B)` with S

= [X, y]: when the right-hand side is not given, the last column of the first argument is split off as y.

Value

A list: x, the solution; residual, the Euclidean norm of $A \% \% x - b$ (NA when A is absent), computed directly in R from A, x and b rather than trusted from TDA's text, since tda_mlsei1 prints no residual at all; rank, the reported rank of A (NA for tda_mnl and tda_mlsei1, neither of which report one).

See Also

Other optimization: [tda_boolean_min\(\)](#), [tda_mlp\(\)](#), [tda_mlpi\(\)](#), [tda_mqap\(\)](#), [tda_mqp\(\)](#)

Examples

```
# unconstrained: fit x1, x2 to two noisy targets
tda_mls(diag(2), c(2.1, 2.9))
# the manual's augmented form S = [X, y] works too
tda_mls(cbind(diag(2), c(2.1, 2.9)))
# x1 + x2 = 3, x1 close to 2, x2 close to 3
tda_mlsei(diag(2), c(2, 3), equalities = rbind(c(1, 1)),
          equalities_bounds = 3)
```

tda_mmul

Matrix multiplication, transposition, Cholesky decomposition

Description

tda_mmul multiplies any number of matrices in order ($A1 \% \% A2 \% \% \dots \% \% Ak$) – TDA's mmul. tda_mtransp transposes – TDA's mtransp. tda_mchol returns the lower-triangular Cholesky factor L such that $L \% \% t(L) == A$ (i.e. $t(\text{chol}(A))$ in base R terms; base R's chol returns the upper factor) – TDA's mchol, requiring a positive-definite A.

Usage

```
tda_mmul(..., dir = tempfile("tda"))

tda_mtransp(A, dir = tempfile("tda"))

tda_mchol(A, dir = tempfile("tda"))
```

Arguments

...	for tda_mmul, two or more numeric matrices to multiply in order.
dir	working directory.
A	a numeric matrix.

Value

A numeric matrix.

See Also

Other matrix algebra: `tda_mcel()`, `tda_mcent()`, `tda_mch()`, `tda_mcross()`, `tda_mdiag()`, `tda_mev()`, `tda_mevs()`, `tda_midf()`, `tda_midf1()`, `tda_midf2()`, `tda_midf3()`, `tda_minvs()`, `tda_mkmet()`, `tda_mkp()`, `tda_mnc()`, `tda_mnrow()`, `tda_mpbl()`, `tda_mpfrit()`, `tda_mpinv()`, `tda_mpit()`, `tda_mple()`, `tda_mpz()`, `tda_mscal1()`, `tda_msqrtd()`, `tda_msvd()`, `tda_mwvec()`

Examples

```
tda_mmul(matrix(1:4, 2), diag(2))
tda_mtransp(matrix(1:6, 2))
tda_mchol(matrix(c(4, 2, 2, 3), 2))
```

 tda_mnc |

 Zero out matrix elements by a threshold test |

Description

Sets $A[i, j]$ to 0 wherever the chosen comparison against x holds, leaving every other element unchanged – TDA's `mnc` (documented in `tda.hlp`). `test` chooses the comparison: " $\leq$ ", " $<$ ", " $\geq$ ", " $>$ " (against the raw value), or the same four prefixed "`abs`" (against `abs(A[i, j])`).

Usage

```
tda_mnc(
  A,
  x,
  test = c("<=", "<", ">=", ">", "abs<=", "abs<", "abs>=", "abs>"),
  dir = tempfile("tda")
)
```

Arguments

<code>A</code>	a numeric matrix.
<code>x</code>	the threshold.
<code>test</code>	which comparison zeroes an element; see above.
<code>dir</code>	working directory.

Value

A numeric matrix.

See Also

Other matrix algebra: `tda_mcel()`, `tda_mcent()`, `tda_mch()`, `tda_mcross()`, `tda_mdiag()`, `tda_mev()`, `tda_mevs()`, `tda_midf()`, `tda_midf1()`, `tda_midf2()`, `tda_midf3()`, `tda_minvs()`, `tda_mkmet()`, `tda_mkp()`, `tda_mmul()`, `tda_mnrow()`, `tda_mpbl()`, `tda_mpfilt()`, `tda_mpinv()`, `tda_mpit()`, `tda_mple()`, `tda_mpz()`, `tda_mscal1()`, `tda_msqrtd()`, `tda_msvd()`, `tda_mwvec()`

Examples

```
A <- matrix(c(1, 5, 3, -2, 4, 6), 2, 3)
tda_mnc(A, 3, "<=") # zero out every element <= 3
```

tda_mnrow	<i>Matrix size and norms</i>
-----------	------------------------------

Description

Scalar summaries of a matrix – TDA's `mnrow`, `mncol`, `mnorm` (max absolute element), `mnorm1` (sum of absolute elements), `mnorm2` (Frobenius/Euclidean norm, $\sqrt{\text{sum}(A^2)}$), and `mtrace` ($\text{sum}(\text{diag}(A))$, using $\min(\text{nrow}(A), \text{ncol}(A))$ diagonal elements for a non-square A , same as base R's $\text{sum}(\text{diag}(A))$). Each returns a plain number, not a 1x1 matrix.

Usage

```
tda_mnrow(A, dir = tempfile("tda"))

tda_mncol(A, dir = tempfile("tda"))

tda_mnorm(A, dir = tempfile("tda"))

tda_mnorm1(A, dir = tempfile("tda"))

tda_mnorm2(A, dir = tempfile("tda"))

tda_mtrace(A, dir = tempfile("tda"))
```

Arguments

<code>A</code>	a numeric matrix.
<code>dir</code>	working directory.

Value

A single number.

See Also

Other matrix algebra: `tda_mcel()`, `tda_mcent()`, `tda_mch()`, `tda_mcross()`, `tda_mdiag()`, `tda_mev()`, `tda_mevs()`, `tda_midf()`, `tda_midf1()`, `tda_midf2()`, `tda_midf3()`, `tda_minvs()`, `tda_mkmet()`, `tda_mkp()`, `tda_mmul()`, `tda_mnc()`, `tda_mpbl()`, `tda_mpfit()`, `tda_mpinv()`, `tda_mpit()`, `tda_mple()`, `tda_mpz()`, `tda_mscal1()`, `tda_msqrtd()`, `tda_msvd()`, `tda_mwvec()`

Examples

```
A <- matrix(c(2, 1, 4, 5, 3, 8, 9, 7, 1), 3)
tda_mnrow(A); tda_mncol(A)
tda_mnorm(A); tda_mnorm1(A); tda_mnorm2(A); tda_mtrace(A)
```

tda_mpbl	<i>Permute to block triangular form</i>
----------	---

Description

Finds a symmetric permutation that puts a square matrix into block lower-triangular (`tda_mpbl`) or block upper-triangular (`tda_mpbu`) form (Duff & Reid, CACM 529) – TDA's `mpbl/mpbu`. `B[i, j] == A[p[i], p[j]]` for every `i, j`; `block` gives, for each row/column of `B`, which block it belongs to (blocks numbered in the order `B` is arranged, so entries between an earlier and a later block are always zero on the side `tda_mpbl/tda_mpbu` promises – confirmed on an instance with cross-block coupling, not one where every block trivially has size 1).

Usage

```
tda_mpbl(A, dir = tempfile("tda"))

tda_mpbu(A, dir = tempfile("tda"))
```

Arguments

<code>A</code>	a square numeric matrix.
<code>dir</code>	working directory.

Value

A list: `B`, the permuted, block-triangular matrix; `p`, the permutation (`B == A[p, p]`); `block`, an integer vector the same length as `p` giving each position's block number.

See Also

Other matrix algebra: `tda_mcel()`, `tda_mcent()`, `tda_mch()`, `tda_mcross()`, `tda_mdiag()`, `tda_mev()`, `tda_mevs()`, `tda_midf()`, `tda_midf1()`, `tda_midf2()`, `tda_midf3()`, `tda_minvs()`, `tda_mkmet()`, `tda_mkp()`, `tda_mmul()`, `tda_mnc()`, `tda_mnrow()`, `tda_mpfit()`, `tda_mpinv()`, `tda_mpit()`, `tda_mple()`, `tda_mpz()`, `tda_mscal1()`, `tda_msqrtd()`, `tda_msvd()`, `tda_mwvec()`

Examples

```
A <- matrix(c(1, 2, 0, 0, 0, 1, 0, 0, 0, 0, 1, 3, 0, 0, 0, 1), 4, 4)
r <- tda_mpbl(A)
r$B
```

tda_mpfit	<i>Iterative proportional fitting</i>
-----------	---------------------------------------

Description

Adjusts a table A so its row and column sums match prescribed targets while preserving cross-product ratios (the RAS/IPF algorithm) – TDA’s mpfit. A must be non-negative with strictly positive row and column sums, as must the target row_sums/col_sums, and their totals should agree (TDA warns but does not refuse otherwise).

Usage

```
tda_mpfit(
  A,
  row_sums,
  col_sums,
  max_iter = 100,
  eps = 1e-06,
  dir = tempfile("tda")
)
```

Arguments

A	a non-negative numeric matrix (n x m).
row_sums	target row sums, length n.
col_sums	target column sums, length m.
max_iter	maximum number of iterations.
eps	convergence tolerance on the maximum deviation between the fitted and target sums.
dir	working directory.

Value

A list: B, the fitted table; iterations, how many were used; accuracy, the final deviation reached.

See Also

Other matrix algebra: [tda_mcel\(\)](#), [tda_mcent\(\)](#), [tda_mch\(\)](#), [tda_mcross\(\)](#), [tda_mdiag\(\)](#), [tda_mev\(\)](#), [tda_mevs\(\)](#), [tda_midf\(\)](#), [tda_midf1\(\)](#), [tda_midf2\(\)](#), [tda_midf3\(\)](#), [tda_minvs\(\)](#), [tda_mkmet\(\)](#), [tda_mkp\(\)](#), [tda_mmul\(\)](#), [tda_mnc\(\)](#), [tda_mnrow\(\)](#), [tda_mpbl\(\)](#), [tda_mpinv\(\)](#), [tda_mpit\(\)](#), [tda_mple\(\)](#), [tda_mpz\(\)](#), [tda_mscal1\(\)](#), [tda_msqrtd\(\)](#), [tda_msvd\(\)](#), [tda_mwvec\(\)](#)

Examples

```
tda_mpfilt(matrix(c(10, 20, 30, 40), 2, 2, byrow = TRUE),
            row_sums = c(45, 55), col_sums = c(40, 60))
```

tda_mpinv	<i>Invert a permutation</i>
-----------	-----------------------------

Description

TDA's mpinv: given a permutation p of $1:n$, returns q such that $q[p[i]] == i$ – matching base R's `order(p)`.

Usage

```
tda_mpinv(p, dir = tempfile("tda"))
```

Arguments

`p` an integer permutation of $1:\text{length}(p)$.
`dir` working directory.

Value

An integer vector, the inverse permutation.

See Also

Other matrix algebra: [tda_mcel\(\)](#), [tda_mcent\(\)](#), [tda_mch\(\)](#), [tda_mcross\(\)](#), [tda_mdiag\(\)](#), [tda_mev\(\)](#), [tda_mevs\(\)](#), [tda_midf\(\)](#), [tda_midf1\(\)](#), [tda_midf2\(\)](#), [tda_midf3\(\)](#), [tda_minvs\(\)](#), [tda_mkmet\(\)](#), [tda_mkp\(\)](#), [tda_mmul\(\)](#), [tda_mnc\(\)](#), [tda_mnrow\(\)](#), [tda_mpbl\(\)](#), [tda_mpfilt\(\)](#), [tda_mpit\(\)](#), [tda_mple\(\)](#), [tda_mpz\(\)](#), [tda_mscal1\(\)](#), [tda_msqrtd\(\)](#), [tda_msvd\(\)](#), [tda_mwvec\(\)](#)

Examples

```
tda_mpinv(c(3, 1, 4, 2))    # == order(c(3, 1, 4, 2))
```

tda_mpit	<i>Iterate a Leslie matrix</i>
----------	--------------------------------

Description

Projects an age-structured population forward using a Leslie matrix – TDA's mpit/mpit1. Row i of fertility_survival holds (fertility[i], survival[i]): fertility[i] is the birth rate contributed by age class i (used only for class 1 of the next generation); survival[i] is the fraction of age class i that survives into age class $i + 1$ (survival[n], the last row, is unused). tda_mpit1 adds a constant vector immigration to every age class after each projection step.

Usage

```
tda_mpit(fertility_survival, population, iterations, dir = tempfile("tda"))

tda_mpit1(
  fertility_survival,
  population,
  immigration,
  iterations,
  dir = tempfile("tda")
)
```

Arguments

fertility_survival	an $n \times 2$ numeric matrix, columns (fertility, survival) as above.
population	the initial population vector, length n .
iterations	how many projection steps to take.
dir	working directory.
immigration	for tda_mpit1, a constant vector added to the population after each step, length n .

Value

A matrix with iterations + 1 rows (the initial population, then one row per step) and n columns.

See Also

Other matrix algebra: [tda_mcel\(\)](#), [tda_mcent\(\)](#), [tda_mch\(\)](#), [tda_mcross\(\)](#), [tda_mdiag\(\)](#), [tda_mev\(\)](#), [tda_mevs\(\)](#), [tda_midf\(\)](#), [tda_midf1\(\)](#), [tda_midf2\(\)](#), [tda_midf3\(\)](#), [tda_minvs\(\)](#), [tda_mkmet\(\)](#), [tda_mkp\(\)](#), [tda_mmul\(\)](#), [tda_mnc\(\)](#), [tda_mnrow\(\)](#), [tda_mtbl\(\)](#), [tda_mpfilt\(\)](#), [tda_mpinv\(\)](#), [tda_mple\(\)](#), [tda_mpz\(\)](#), [tda_mscal1\(\)](#), [tda_msqrtd\(\)](#), [tda_msvd\(\)](#), [tda_mwvec\(\)](#)

Examples

```
F <- rbind(c(0, 0), c(2, 0.5), c(1, 0.3))
tda_mpit(F, c(100, 50, 20), iterations = 3)
```

tda_mple

Product-limit (Kaplan-Meier) distribution and jumps

Description

The product-limit (Kaplan-Meier) estimate of the CDF for a vector of possibly right-censored event times – TDA’s `mple`, via Efron’s redistribute-to-the-right algorithm, applied one observation at a time in sorted order (events before censoring at the same time). When no two events fall at exactly the same time, this matches the standard product-limit CDF exactly – confirmed against `survival::survfit` across random instances (events may still tie with a censored observation; only ties *among events* are excluded). When two or more events tie, processing them one at a time rather than as a single simultaneous risk-set reduction means only the *last* observation processed within that tied group reaches the value standard Kaplan-Meier software reports for that time – earlier ones in the same group show smaller, order-dependent intermediate values. Two events tied at time = 2 out of 3 total observations give $F = (1/3, 2/3, 1)$ here, where standard Kaplan-Meier reports $F(2) = 2/3$ for both tied observations. This is a property of the algorithm as TDA implements it, not a bug and not this wrapper’s choice to make. Separately: F always reaches exactly 1 at the highest-time observation, *even when that observation is censored* – redistribution only moves probability mass to later observations, never destroys it, and the highest-time observation has nothing later to redistribute its share to, so all of the mass ends up accounted for there by construction. Standard Kaplan-Meier software instead leaves the survival curve at whatever value it last reached, undefined beyond a censored tail. `time = c(1, 2, 3)`, `censored = c(0, 0, 1)` gives $F = (1/3, 2/3, 1)$ here. This is the low-level building block behind TDA’s `tda_ple`, which most users want instead – this one works directly on plain vectors, with no grouping, formula interface, or standard errors.

Usage

```
tda_mple(time, censored, dir = tempfile("tda"))
```

Arguments

<code>time</code>	a numeric vector of event/censoring times.
<code>censored</code>	a numeric vector the same length as <code>time</code> : 1 where the observation is censored, 0 where it is an observed event.
<code>dir</code>	working directory.

Value

A list, both the same length as `time`, in the original (unsorted) order: F , the estimated CDF at each observation; `jumps`, the increase in F at each observation (0 for a censored observation).

See Also

Other matrix algebra: [tda_mcel\(\)](#), [tda_mcent\(\)](#), [tda_mch\(\)](#), [tda_mcross\(\)](#), [tda_mdiag\(\)](#), [tda_mev\(\)](#), [tda_mevs\(\)](#), [tda_midf\(\)](#), [tda_midf1\(\)](#), [tda_midf2\(\)](#), [tda_midf3\(\)](#), [tda_minvs\(\)](#), [tda_mkmet\(\)](#), [tda_mkp\(\)](#), [tda_mmul\(\)](#), [tda_mnc\(\)](#), [tda_mnrow\(\)](#), [tda_mpbl\(\)](#), [tda_mpfit\(\)](#), [tda_mpinv\(\)](#), [tda_mpit\(\)](#), [tda_mpz\(\)](#), [tda_mscal1\(\)](#), [tda_msqrtd\(\)](#), [tda_msvd\(\)](#), [tda_mwvec\(\)](#)

Examples

```
# six durations, two of them censored (still running at 3 and 7)
t <- c(2, 3, 3, 5, 7, 8)
cen <- c(0, 0, 1, 0, 1, 0)
pl <- tda_mple(t, cen)
pl$F      # the Kaplan-Meier distribution function at each observation
pl$jumps  # its increase there; 0 at the censored ones
```

tda_mproc	<i>Procrustes rotation</i>
-----------	----------------------------

Description

TDA's mproc: rotates configuration Y to best match X. Verified in the suite against the closed-form solution from the singular value decomposition.

Usage

```
tda_mproc(x, y, ...)
```

Arguments

- x, y n-by-m matrices.
- ... passed to [tda_run](#).

Value

the rotated configuration as a matrix.

Examples

```
x <- cbind(c(0, 1, 0), c(0, 0, 1))
th <- 0.7; R <- rbind(c(cos(th), -sin(th)), c(sin(th), cos(th)))
round(tda_mproc(x, x %%% R), 6)
```

tda_mpz

*Permute rows for a zero-free diagonal***Description**

Finds a row permutation that gives a square matrix a zero-free diagonal wherever one exists (Duff's algorithm, ACM 575) – TDA's mpz. p satisfies $A[p[i], i] \neq 0$ for every i (the exact contract `m_rperm`'s header states), and B is built as the gather $B[i,] == A[p[i],]$, so $\text{diag}(B)[i] == A[p[i], i]$ by construction. An earlier version of TDA's C code built B as the opposite (a scatter, $B[p[i],] == A[i,]$) despite p itself already satisfying the gather contract – confirmed wrong on a 3×3 matrix with a unique, forced perfect matching (a pure permutation matrix, one nonzero per row and column): the scatter form returned an all-zero diagonal even though TDA's diagnostic confirmed a complete matching was found. Fixed at the source (`m_mpz`, `t_matc.c`), not worked around here. If A is structurally singular, some diagonal entries of B may still be zero – check `all(diag(B) != 0)` if that matters, rather than assuming a full match always exists.

Usage

```
tda_mpz(A, dir = tempfile("tda"))
```

Arguments

`A` a square numeric matrix.
`dir` working directory.

Value

A list: B , the permuted matrix; p , the permutation, with $A[p[i], i] \neq 0$ for every i where that is achievable.

See Also

Other matrix algebra: `tda_mcel()`, `tda_mcent()`, `tda_mch()`, `tda_mcross()`, `tda_mdiag()`, `tda_mev()`, `tda_mevs()`, `tda_midf()`, `tda_midf1()`, `tda_midf2()`, `tda_midf3()`, `tda_minvs()`, `tda_mkmet()`, `tda_mkp()`, `tda_mmul()`, `tda_mnc()`, `tda_mnrow()`, `tda_mpb1()`, `tda_mpf1t()`, `tda_mpinv()`, `tda_mpit()`, `tda_mple()`, `tda_mscal1()`, `tda_msqrtd()`, `tda_msvd()`, `tda_mwvec()`

Examples

```
A <- matrix(c(0, 3, 0, 1, 0, 5, 2, 4, 0), 3, 3)
r <- tda_mpz(A)
r$p
all(diag(r$B) != 0)
```

tda_mqap

*Quadratic assignment problem***Description**

Approximate solution of the quadratic assignment problem – TDA's mqap, adapted from CACM algorithm 608 (D.H. West). Tries to find a permutation p of $1:n$ minimizing

$$\sum_i \left(C_{i,p(i)} + \sum_j F_{ij} D_{p(i),p(j)} \right)$$

This is a heuristic, not an exact solver: the value returned is an upper bound on the true minimum, and need not be optimal for larger n (see `examples/coverage/mqapops.cf` for the objective and small hand-verified instances). `flows` and `distances` must have a zero main diagonal; a nonzero diagonal is silently reset to zero with a warning from TDA itself.

Usage

```
tda_mqap(flows, distances, costs = NULL, dir = tempfile("tda"), ...)
```

Arguments

<code>flows</code>	the flow matrix F ($n \times n$).
<code>distances</code>	the distance matrix D ($n \times n$).
<code>costs</code>	the fixed placement cost matrix C ($n \times n$), default all zero.
<code>dir</code>	working directory.
<code>...</code>	passed to tda_run .

Value

A list: value, the objective at the returned permutation; permutation, integer vector p with $p[i]$ the location assigned to facility i .

See Also

Other optimization: [tda_boolean_min\(\)](#), [tda_mlp\(\)](#), [tda_mlpi\(\)](#), [tda_mls\(\)](#), [tda_mqp\(\)](#)

Examples

```
F <- matrix(c(0,5,2,4, 5,0,3,0, 2,3,0,0, 4,0,0,0), 4, 4, byrow = TRUE)
D <- matrix(c(0,8,15,13, 8,0,9,10, 15,9,0,17, 13,10,17,0), 4, 4,
            byrow = TRUE)
tda_mqap(F, D)
```

tda_mqp

*Quadratic programming***Description**

Minimize a convex quadratic subject to box bounds and/or linear equality/inequality constraints – TDA’s mqp, mqpb, and mqpc (undocumented in `tda.hlp`; identified from `qld`’s header, the Powell/Schittkowski active-set solver ZQPCVX shared with [tda_mlsei1](#)). The problem solved is

$$\min d'x + \frac{1}{2}x'Cx \quad \text{s.t. } A_ex = b_e, A_ix \geq b_i, l \leq x \leq u$$

`tda_mqp` is unconstrained (box bounds fixed at +-huge internally by TDA); `tda_mqpb` adds box bounds; `tda_mqpc` adds general equality/inequality constraints instead of bounds – no single TDA command accepts both bounds and general constraints. The suite checks all three commands against hand-solved instances: an unconstrained case, a box-clipped case, and a mixed equality+inequality case – see `examples/coverage/mqpops2.cf`.

Usage

```
tda_mqp(C, d, dir = tempfile("tda"), ...)

tda_mqpb(C, d, lower, upper, dir = tempfile("tda"), ...)

tda_mqpc(
  C,
  d,
  equalities = NULL,
  equalities_bounds = NULL,
  inequalities = NULL,
  inequalities_bounds = NULL,
  dir = tempfile("tda"),
  ...
)
```

Arguments

<code>C</code>	the symmetric objective matrix (n x n).
<code>d</code>	the linear objective term, length n.
<code>dir</code>	working directory.
<code>...</code>	passed to tda_run .
<code>lower, upper</code>	box bounds, each length n. <code>tda_mqpb</code> only.
<code>equalities</code>	an equality constraint matrix A_e $x = equalities_bounds$. <code>tda_mqpc</code> only.
<code>equalities_bounds</code>	the right-hand sides for equalities.
<code>inequalities</code>	an inequality constraint matrix A_i $x \geq inequalities_bounds$. <code>tda_mqpc</code> only.
<code>inequalities_bounds</code>	the right-hand sides for inequalities.

Value

A list: `x`, the solution; `value`, the objective $d'x + 0.5 x'Cx$ at `x`, computed directly in R since this command family prints no value of its own.

See Also

Other optimization: [tda_boolean_min\(\)](#), [tda_mlp\(\)](#), [tda_mlpi\(\)](#), [tda_mls\(\)](#), [tda_mqap\(\)](#)

Examples

```
# minimize  $x_1^2 + x_2^2 - 2x_1 - 4x_2$  (unconstrained optimum (1, 2))
tda_mqp(diag(c(2, 2)), c(-2, -4))
tda_mqpb(diag(c(2, 2)), c(-2, -4), lower = c(0, 0), upper = c(0.5, 0.5))
```

tda_mreg	<i>Monotone regression</i>
----------	----------------------------

Description

Rather than assume the response is already on an interval scale, `mreg` finds the monotone rescaling of it that fits a linear model best – Kruskal’s non-metric approach, the same idea non-metric MDS uses. It reports a coefficient table but no standard errors: `mreg` does not compute them (no `pcov=` in its own TDA implementation), so `vcov()` and `confint()` are always NA here, honestly reflecting what TDA itself reports rather than a parsing gap.

Usage

```
tda_mreg(
  formula,
  data,
  ties = c("ignore", "primary", "secondary"),
  control = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

<code>formula</code>	a two-sided formula.
<code>data</code>	a data frame.
<code>ties</code>	how to handle tied response values: "ignore" (default), or Kruskal’s "primary" or "secondary" approach to them; <code>mreg</code> ’s <code>opt=</code> . On data with ties, "ignore" and "secondary" converge to a different final stress, not the same value under two names.
<code>control</code>	convergence settings from tda_control (<code>mxit=/tolf=/tolsp=</code>).

options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to <code>tda_run</code> .

Details

`predict()` (and `fitted()`, its default with no `newdata`) is well defined here – the coefficients describe a linear model, just of the rescaled response rather than the original one – but the numbers it returns are on that rescaled (“disparity”) scale, not the original response’s units or levels. This is the same idea as `predict.glm(type = "link")` returning the linear predictor rather than a probability.

Value

An object of class `tda_fit`.

See Also

Other regression: [TDA_FAMILIES](#), [TDA_QRMODELS](#), [tda_freq\(\)](#), [tda_gdf\(\)](#), [tda_l1reg\(\)](#), [tda_lsreg\(\)](#), [tda_mlrc_design\(\)](#), [tda_nlreg\(\)](#), [tda_npreg\(\)](#), [tda_zreg\(\)](#)

Examples

```
set.seed(52)
d <- data.frame(x = round(rnorm(50), 2))
# y is an ordinal rank (1-5), not an interval scale, driven by a linear
# score plus noise -- exactly the case mreg is meant for
score <- 1 + 2 * d$x + rnorm(50, sd = 0.4)
d$y <- as.integer(cut(score, quantile(score, seq(0, 1, 0.2)),
                      include.lowest = TRUE))
fit <- tda_mreg(y ~ x, d)
coef(fit)

# y is heavily tied by construction (5 ordinal ranks) -- ties= selects
# how mreg's algorithm resolves that; see Details for a directly
# verified case where it changes the fit's stress
coef(tda_mreg(y ~ x, d, ties = "secondary"))

# fitted values, on the rescaled response -- not the original 1-5 scale,
# but increasing with x the way the underlying score was
predict(fit)
predict(fit, newdata = data.frame(x = c(-1, 0, 1)))
```

Description

tda_mrsum/tda_mcsun are TDA's mrsum/mcsun, matching base R's rowSums/colSums. tda_mdrow/tda_mdcol return the same sums as a diagonal matrix instead of a vector – TDA's mdrow/ mdcol, matching `diag(rowSums(A))/ diag(colSums(A))`; useful for building a graph's degree matrix from its adjacency matrix.

Usage

```
tda_mrsum(A, dir = tempfile("tda"))
tda_mcsun(A, dir = tempfile("tda"))
tda_mdrow(A, dir = tempfile("tda"))
tda_mdcol(A, dir = tempfile("tda"))
```

Arguments

A a numeric matrix.
 dir working directory.

Value

tda_mrsum/tda_mcsun return a plain numeric vector; tda_mdrow/tda_mdcol return a diagonal matrix.

See Also

Other matrix reshaping: [tda_mag\(\)](#), [tda_mcat\(\)](#), [tda_mvec\(\)](#), [tda_msort\(\)](#), [tda_msrow\(\)](#), [tda_mtrim\(\)](#)

Examples

```
A <- matrix(1:6, 2, 3)
tda_mrsum(A); tda_mcsun(A)
tda_mdrow(A); tda_mdcol(A)
```

tda_mscall	<i>Scale a matrix by its total sum</i>
------------	--

Description

Divides every element of A by the sum of all its elements – TDA's mscall, matching `A / sum(A)`.

Usage

```
tda_mscall(A, dir = tempfile("tda"))
```

Arguments

A a numeric matrix whose elements do not sum to zero.
 dir working directory.

Value

A numeric matrix.

See Also

Other matrix algebra: [tda_mcel\(\)](#), [tda_mcent\(\)](#), [tda_mch\(\)](#), [tda_mcross\(\)](#), [tda_mdiag\(\)](#), [tda_mev\(\)](#), [tda_mevs\(\)](#), [tda_midf\(\)](#), [tda_midf1\(\)](#), [tda_midf2\(\)](#), [tda_midf3\(\)](#), [tda_minvs\(\)](#), [tda_mkmet\(\)](#), [tda_mkp\(\)](#), [tda_mmul\(\)](#), [tda_mnc\(\)](#), [tda_mnrow\(\)](#), [tda_mpbl\(\)](#), [tda_mpfilt\(\)](#), [tda_mpinv\(\)](#), [tda_mpit\(\)](#), [tda_mple\(\)](#), [tda_mpz\(\)](#), [tda_msqrtd\(\)](#), [tda_msvd\(\)](#), [tda_mwvec\(\)](#)

Examples

```
A <- matrix(c(1, 2, 3, 4), 2)
tda_mscal1(A) # A / sum(A): the cells now sum to one
```

tda_msort	<i>Sort or rank the rows of a matrix</i>
-----------	--

Description

tda_msort sorts the rows of X ascending by the columns named in by, first column first – TDA's msort, matching `X[do.call(order, as.data.frame(X[, by])), ]`. tda_msort1 does the same and then drops consecutive duplicate rows (comparing every column, not just by) – TDA's msort1, matching `unique(sorted_X)`. tda_mrank – despite the name – is *not* a linear-algebra matrix rank (no TDA command returns one as a matrix value; [tda_mginv](#) prints a pseudorank as a side effect, but never returns it). It returns the sort *permutation* itself: an integer vector p such that `X[p, ]` is sorted by by – TDA's mrank, which equals base R's `order(X[, by[1]], X[, by[2]], ...)`, not `rank()`.

Usage

```
tda_msort(X, by, dir = tempfile("tda"))

tda_msort1(X, by, dir = tempfile("tda"))

tda_mrank(X, by, dir = tempfile("tda"))
```

Arguments

X a numeric matrix.
 by which columns to sort by, in order (1-based, at least one).
 dir working directory.

Value

tda_msort/tda_msort1 return a matrix; tda_mrank returns an integer vector, the same length as nrow(X).

See Also

Other matrix reshaping: [tda_mag\(\)](#), [tda_mcat\(\)](#), [tda_mcvec\(\)](#), [tda_mrsum\(\)](#), [tda_msrow\(\)](#), [tda_mtrim\(\)](#)

Examples

```
X <- rbind(c(30, 1), c(10, 1), c(20, 1), c(40, 1))
tda_msort(X, 1)
tda_mrank(X, 1) # == order(X[, 1]), i.e. c(2, 3, 1, 4)
```

tda_msqrt	<i>Diagonal square root and its reciprocal</i>
-----------	--

Description

Given a square matrix A, returns a diagonal matrix (zero off-diagonal) built from its diagonal – TDA's msqrt(diag(sqrt(diag(A)))); refuses a negative diagonal element) and msqrti (diag(1/sqrt(diag(A)))); refuses an (almost) zero diagonal element). Off-diagonal entries of A are ignored entirely, same as [tda_minvd](#).

Usage

```
tda_msqrt(A, dir = tempfile("tda"))

tda_msqrti(A, dir = tempfile("tda"))
```

Arguments

A a square numeric matrix.
 dir working directory.

Value

A diagonal numeric matrix.

See Also

Other matrix algebra: [tda_mcel\(\)](#), [tda_mcent\(\)](#), [tda_mch\(\)](#), [tda_mcross\(\)](#), [tda_mdiag\(\)](#), [tda_mev\(\)](#), [tda_mevs\(\)](#), [tda_midf\(\)](#), [tda_midf1\(\)](#), [tda_midf2\(\)](#), [tda_midf3\(\)](#), [tda_minvs\(\)](#), [tda_mkmet\(\)](#), [tda_mkp\(\)](#), [tda_mmul\(\)](#), [tda_mnc\(\)](#), [tda_mnrow\(\)](#), [tda_mpbl\(\)](#), [tda_mpfit\(\)](#), [tda_mpinv\(\)](#), [tda_mpit\(\)](#), [tda_mple\(\)](#), [tda_mpz\(\)](#), [tda_mscal1\(\)](#), [tda_msvd\(\)](#), [tda_mwvec\(\)](#)

Examples

```
tda_msqrt(matrix(c(4, 1, 1, 9), 2))    # diag(2, 3)
tda_msqrti(matrix(c(4, 1, 1, 9), 2))   # diag(0.5, 1/3)
```

tda_msrow	<i>Selecting or permuting rows and columns</i>
-----------	--

Description

tda_msrow/tda_mscol select (and may reorder or repeat) rows/columns by index – TDA's msrow/mscol, matching `X[rows, ]/X[, cols]`. tda_mprow/ tda_mpcol/tda_mpsym permute rows, columns, or both by the same permutation – TDA's mprow/mpcol/ mpsym, matching `A[p, ]/A[, p]/A[p, p]` (m_mperm's header states the convention, $b(i, j) = a(p(i), j)$ etc.). mprow/mpcol require a permutation of $1:nrow(A)/1:ncol(A)$, so unlike msrow/ mscol they cannot repeat or drop indices; mpsym additionally requires A to be square.

Usage

```
tda_msrow(X, rows, dir = tempfile("tda"))

tda_mscol(X, cols, dir = tempfile("tda"))

tda_mprow(A, p, dir = tempfile("tda"))

tda_mpcol(A, p, dir = tempfile("tda"))

tda_mpsym(A, p, dir = tempfile("tda"))
```

Arguments

X, A	a numeric matrix.
rows, cols	integer indices (1-based; tda_msrow/ tda_mscol allow repeats and omissions).
dir	working directory.
p	an integer permutation of $1:nrow(A)$ (tda_mprow, tda_mpsym) or $1:ncol(A)$ (tda_mpcol).

Value

A numeric matrix.

See Also

Other matrix reshaping: [tda_mag\(\)](#), [tda_mcat\(\)](#), [tda_mcvec\(\)](#), [tda_mrsum\(\)](#), [tda_msort\(\)](#), [tda_mtrim\(\)](#)

Examples

```
A <- matrix(1:9, 3, 3)
tda_msrow(A, c(2, 1))      # == A[c(2, 1), ]
tda_mprow(A, c(2, 3, 1))   # == A[c(2, 3, 1), ]
tda_mpsym(A, c(2, 3, 1))   # == A[c(2, 3, 1), c(2, 3, 1)]
```

tda_msvd

Singular value decomposition

Description

tda_msvd returns just the singular values – TDA’s msvd. tda_msvd1 returns the full decomposition $A = U \%*\% \text{diag}(d) \%*\% t(V)$ – TDA’s msvd1, matching base R’s svd() convention exactly (singular vectors as columns of u/v). A must have at least as many rows as columns for either. As with any singular vector, each column’s sign is arbitrary and need not match svd()’s choice.

Usage

```
tda_msvd(A, dir = tempfile("tda"))

tda_msvd1(A, dir = tempfile("tda"))
```

Arguments

A	a numeric matrix with at least as many rows as columns.
dir	working directory.

Value

tda_msvd returns a plain numeric vector of singular values. tda_msvd1 returns a list: d, the singular values; u, v, the singular vectors as columns.

See Also

Other matrix algebra: [tda_mcel\(\)](#), [tda_mcent\(\)](#), [tda_mch\(\)](#), [tda_mcross\(\)](#), [tda_mdiag\(\)](#), [tda_mev\(\)](#), [tda_mevs\(\)](#), [tda_midf\(\)](#), [tda_midf1\(\)](#), [tda_midf2\(\)](#), [tda_midf3\(\)](#), [tda_minvs\(\)](#), [tda_mkmet\(\)](#), [tda_mkp\(\)](#), [tda_mmul\(\)](#), [tda_mnc\(\)](#), [tda_mnrow\(\)](#), [tda_mpbl\(\)](#), [tda_mpfit\(\)](#), [tda_mpinv\(\)](#), [tda_mpit\(\)](#), [tda_mple\(\)](#), [tda_mpz\(\)](#), [tda_mscal1\(\)](#), [tda_msqrtd\(\)](#), [tda_mwvec\(\)](#)

Examples

```
A <- matrix(c(1, 3, 5, 2, 4, 7), 3, 2)
tda_msvd(A)      # == svd(A)$d
tda_msvd1(A)     # == svd(A), up to per-column sign
```

tda_mtrim

*Trim or pad a matrix's edges***Description**

Deletes or adds whole rows/columns at the edges of A – TDA's mtrim. A positive count deletes that many rows or columns from that edge; a negative count adds that many zero rows/columns instead. leading_cols/trailing_cols act on the first and last columns, leading_rows/ trailing_rows on the first and last rows.

Usage

```
tda_mtrim(
  A,
  leading_cols = 0,
  leading_rows = 0,
  trailing_cols = 0,
  trailing_rows = 0,
  dir = tempfile("tda")
)
```

Arguments

A a numeric matrix.
 leading_cols, trailing_cols, leading_rows, trailing_rows
 integer counts (default 0); positive deletes, negative pads with zeros.
 dir working directory.

Value

A numeric matrix.

See Also

Other matrix reshaping: [tda_mag\(\)](#), [tda_mcat\(\)](#), [tda_mcvec\(\)](#), [tda_mrsum\(\)](#), [tda_msort\(\)](#), [tda_msrow\(\)](#)

Examples

```
A <- matrix(1:4, 2)
tda_mtrim(A, leading_cols = 1) # drop the first column
tda_mtrim(A, leading_cols = -1) # add a zero column in front
```

tda_mwvec

*Weighted mean of the later elements***Description**

For each position i , averages $a[j]$ weighted by $weights[j]$ over every later position (tda_mwvec: every $j > i$; tda_mwvec1: every j with $t[j] > t[i]$, letting an arbitrary order vector t stand in for position – tda_mwvec is the special case $t = seq_along(a)$) – TDA's mwvec/mwvec1. Falls back to $a[i]$ unchanged wherever the weights beyond i sum to zero (including the last position, which always has nothing after it). No base-R builtin matches this directly.

Usage

```
tda_mwvec(a, weights, dir = tempfile("tda"))
```

```
tda_mwvec1(a, weights, t, dir = tempfile("tda"))
```

Arguments

<code>a, weights</code>	numeric vectors of the same length.
<code>dir</code>	working directory.
<code>t</code>	for tda_mwvec1, a numeric order vector the same length as <code>a</code> ; ties are included on neither side (only strictly later positions count).

Value

A numeric vector, the same length as `a`.

See Also

Other matrix algebra: [tda_mcel\(\)](#), [tda_mcent\(\)](#), [tda_mch\(\)](#), [tda_mcross\(\)](#), [tda_mdiag\(\)](#), [tda_mev\(\)](#), [tda_mevs\(\)](#), [tda_midf\(\)](#), [tda_midf1\(\)](#), [tda_midf2\(\)](#), [tda_midf3\(\)](#), [tda_minvs\(\)](#), [tda_mkmet\(\)](#), [tda_mkp\(\)](#), [tda_mmul\(\)](#), [tda_mnc\(\)](#), [tda_mnrow\(\)](#), [tda_mpbl\(\)](#), [tda_mpfilt\(\)](#), [tda_mpinv\(\)](#), [tda_mpit\(\)](#), [tda_mple\(\)](#), [tda_mpz\(\)](#), [tda_mscal1\(\)](#), [tda_msqrtd\(\)](#), [tda_msvd\(\)](#)

Examples

```
a <- c(10, 20, 30, 40)
w <- c(1, 1, 1, 1)
# position 1 averages 20, 30, 40; position 3 averages 40 alone;
# the last position has nothing after it and keeps its own value
tda_mwvec(a, w)
# with an order vector, "later" means a larger t: positions 1 and 2
# share t = 1, so both average the values at t = 2 and t = 3
tda_mwvec1(a, w, t = c(1, 1, 2, 3))
```

tda_ndvar	<i>Dummy variables</i>
-----------	------------------------

Description

TDA's ndvar: expands variables into dummies. Verified in the suite against `model.matrix`.

Usage

```
tda_ndvar(data, ...)
```

Arguments

<code>data</code>	data frame of integer-valued variables.
<code>...</code>	passed to <code>tda_run</code> .

Value

the printed output describing the created dummies.

Examples

```
# one dummy per observed level of edu, named edu_1, edu_2, edu_3
tda_ndvar(data.frame(edu = c(1, 2, 3, 2, 1)))
```

tda_niset	<i>Numerical-integration settings</i>
-----------	---------------------------------------

Description

TDA's niset: sets tolerance and subdivision options used by the `int` operator within the same run; exposed for pipeline use.

Usage

```
tda_niset(rel_error = 1e-04, method = 1, commands = character(), ...)
```

Arguments

<code>rel_error</code>	relative error target for the integrator.
<code>method</code>	integration algorithm 1-5 (TDA's rhs of niset), default 1, QNG.
<code>commands</code>	further command lines to run under the setting.
<code>...</code>	passed to <code>tda_run</code> .

Value

the printed output, invisibly.

Examples

```
out <- tda_niset(rel_error = 1e-6,
               commands = "int(ab=0,1, fmt=12.8) = sin(x);")
grep("Approximation", out, value = TRUE) # 1 - cos(1)
```

tda_npreg	<i>Non-parametric regression</i>
-----------	----------------------------------

Description

tda_npreg is a non-parametric regression evaluated at points you choose: TDA's npreg, a kernel-smoothed mean, quantile or frequency.

Usage

```
tda_npreg(
  formula,
  data,
  x,
  method = c("mean", "quantile", "frequency", "lowess", "midmeans"),
  kernel = c("uniform", "triangle", "quartic", "epanechnikov"),
  bandwidth = NULL,
  select = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

formula	a two-sided formula.
data	a data frame.
x	for tda_npreg, the points at which to evaluate the fit.
method	see Details; npreg's opt=. Either a name (partially matched) or TDA's number directly (opt=1 is "mean", and so on) – useful when translating a .cf file, which only ever writes the number.
kernel	the smoothing kernel, only meaningful for method = "mean" – npreg's k=. A name or TDA's number, the same as method.
bandwidth	the kernel bandwidth (npreg's d=, default 1).
select	a case-selection expression, TDA's sel=.
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to tda_run .

Details

method picks which: "mean" (the default), a kernel-smoothed mean at each point in x ; "quantile", smoothed quantiles; "frequency", smoothed frequencies. Only these three exist. npreg names the columns for method "mean" and "quantile", which tda_npreg reproduces; "frequency" writes columns that depend on the data (the distinct response categories) and are returned as TDA numbered them, V1, V2, ...

Value

An object carrying the fitted values as table.

See Also

Other regression: [TDA_FAMILIES](#), [TDA_QRMODELS](#), [tda_freg\(\)](#), [tda_gdf\(\)](#), [tda_l1reg\(\)](#), [tda_lsreg\(\)](#), [tda_mlrc_design\(\)](#), [tda_mreg\(\)](#), [tda_nlreg\(\)](#), [tda_zreg\(\)](#)

Examples

```
set.seed(19)
d <- data.frame(x = round(rnorm(60), 3))
d$y <- round(1 + sin(d$x) + rnorm(60, sd = 0.2), 3)
np <- tda_npreg(y ~ x, d, x = seq(-2, 2, 0.5))
np$table

# a triangle kernel, wider bandwidth
np2 <- tda_npreg(y ~ x, d, x = seq(-2, 2, 0.5), kernel = "triangle",
                 bandwidth = 0.5)
np2$table
```

tda_options

TDA options not exposed as named arguments

Description

Most wrappers accept an `options = list(...)` argument that is passed straight through to the TDA command they build. TDA has several hundred options and promoting each to a named R argument would be unreadable, so the named arguments cover what is needed in ordinary use and `options=` reaches the rest.

Which options a command accepts

TDA's manual is installed with the package and is the authoritative list. [tda_help](#) prints the section for any command, including its full option table:

```
tda_help("rate")      # every option the rate command takes
tda_help("sdgshhs")   # the GSHHS reader's options
```

The name in `options=` is TDA's name, exactly as that table spells it, so `options = list(grp = "SEX")` becomes `grp = SEX`, in the generated command.

options= and ... are not the same thing

Where a wrapper has both, options= goes to the *TDA command* and ... goes to `tda_run` (working directory, echo, and so on). Passing a TDA option in ... is therefore an error rather than a silent no-op:

```
tda_rate(..., grp = "SEX")           # unused argument
tda_rate(..., options = list(grp = "SEX")) # reaches the command
```

A few readers – `tda_read_gshhs` and the other spatial readers among them – have no separate options= and forward ... into the command directly, so for those `tda_read_gshhs(f, opt = 2)` is right.

Caveat

An option reaching the command is not the same as the option working. TDA validates its parameters and some are only meaningful for particular models: `grp=` on rate, for instance, reaches the command and is accepted, but the fit then returns nothing unless the model is one of the stratified ones. Check the run's output when an option seems to have no effect.

Examples

```
# what does the GSHHS reader accept?
tda_help("sdgshhs")

# opt = 2 keeps longitudes on 0..360, which stops dateline-crossing
# polygons being drawn straight across a world map
## Not run:
tda_read_gshhs("gshhs_i.b", opt = 2, level = 1)

## End(Not run)
```

 tda_output | *Estimates and raw output* |**Description**

`tda_estimates` returns the coefficient table, reading the column layout from TDA's header so that it works for the rate models (SN/Org/Des/MT), `qreg` (Cat/Term) and `fml` (Parameter/Value) alike. A parameter TDA held fixed, printed as ---, comes back NA. `tda_output` prints everything TDA wrote.

Usage

```
tda_output(x)

tda_estimates(x)
```

Arguments

x a run or fitted model.

Value

A data frame, a list of data frames, or NULL.

See Also

Other TDA infrastructure: [tda_ccnt\(\)](#), [tda_help\(\)](#), [tda_read_table\(\)](#), [tda_run\(\)](#), [tda_write_data\(\)](#)

Examples

```
set.seed(38)
d <- data.frame(x = rnorm(100))
d$y <- rbinom(100, 1, plogis(0.2 + 0.7 * d$x))
f <- tda_qreg(y ~ x, d)
tda_estimates(f)
tda_output(f) # everything TDA printed, not just the coefficient table
```

tda_output_audit	<i>Account for every line of a run's printed output</i>
------------------	---

Description

Classifies each line of output as blank, boilerplate, an echo of the inputs, an optimiser-settings line, content the fitted object stores, content also delivered through the direct-export channel, or *unaccounted*. The last category is the point: it lists what would be lost if the printed output were ever discarded, and the package's tests require it to be empty for a corpus of fits.

Usage

```
tda_output_audit(x)
```

Arguments

x a `tda_result` (from [tda_run](#)) or any fitted object carrying one as `$run`.

Value

a data frame with columns `line` and `class`, one row per output line, with a `summary` attribute counting each class.

Examples

```
d <- data.frame(x = 1:8, y = (1:8) * 2 + rnorm(8))
f <- tda_lsreg(y ~ x, d)
a <- tda_output_audit(f)
attr(a, "summary")
subset(a, class == "UNACCOUNTED")
```

tda_payload	<i>Extract a command's result section from TDA output</i>
-------------	---

Description

TDA's protocol is structured: a header, then each command's echo followed by its result, with separator lines between commands. `tda_payload` returns the result section of the last command (or the n -th from the end), without the surrounding protocol, which is what an example or a quick look usually wants.

Usage

```
tda_payload(output, n = 1L)
```

Arguments

<code>output</code>	a character vector of protocol lines, or a <code>tda_run</code> result (its <code>\$output</code> is used).
<code>n</code>	which command's section, counted from the end; default 1, the last command.

Value

the lines of that section, invisibly empty if the output has no separator structure.

Examples

```
r <- tda_run("int(ab=0,1, fmt=12.8) = x*x;")
cat(tda_payload(r), sep = "\n") # the integral, 1/3
```

tda_pcyd	<i>Cycle structure of permutations</i>
----------	--

Description

TDA's `pcyc`: each record of the given integer variables is a permutation; the command reports its cycle decomposition. Verified in the suite against a direct R cycle count.

Usage

```
tda_pcyd(perm, ...)
```

Arguments

<code>perm</code>	a matrix or data frame, one permutation per row.
<code>...</code>	passed to <code>tda_run</code> .

Value

the output file's lines (one cycle report per record).

Examples

```
# each row is a permutation of 1..4, written as where each element
# goes; the last column of the output is its cycle decomposition
tda_pcyrc(rbind(c(2, 3, 1, 4),    # (1,2,3)(4)
                c(1, 2, 3, 4),    # the identity, four fixed points
                c(2, 1, 4, 3)))   # (1,2)(3,4)
```

tda_pdatd

*Distance matrix from raw variables***Description**

pdatd: builds a case-by-case distance matrix directly from a data matrix, the way [dist](#) does – Euclidean or city-block distance, the count of variables on which two cases differ, or a dissimilarity index. The result feeds directly into [tda_cluster](#) or [tda_mds](#). Verified against [dist](#) for Euclidean and city-block (Manhattan) distance: exact match.

Usage

```
tda_pdatd(
  data,
  measure = 1,
  variables = NULL,
  weights = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

data	a data frame or matrix, one row per case.
measure	the distance measure: 1 Euclidean (default), 2 city-block, 3 number of variables on which the two cases differ, 4 a dissimilarity index.
variables	optional subset of data's column names to use, instead of all of them.
weights	optional positive per-variable weights, one per (selected) variable, all distinct – see Details for TDA's parser constraint and the exact weighting convention.
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to tda_run . opt= (the command's variant switch) passes through as well.

Details

As with `tda_dma`, TDA's `v=` needs variable names that start uppercase to be referenced correctly; `variables` handles this automatically.

Per-variable weights work through `weights`, with one genuine TDA quirk absorbed for you: `wt=` is parsed by TDA's *time-points* parser, which accepts only a strictly increasing positive list – `wt=2,1,1` is a syntax error while `wt=1,2,4` is fine, which is why this option long looked dead. Since a distance does not care about variable order, the wrapper sends the variables sorted by ascending weight. The weighting convention, verified against R directly: euclidean uses $d^2 = \sum_j w_j \Delta_j^2$ (so it matches `dist(sweep(x, 2, sqrt(w), "*"))`), city-block $d = \sum_j w_j |\Delta_j|$. *Tied* weights cannot pass TDA's parser at all; scale those columns yourself ($\sqrt{wx}$ for euclidean, wx for city-block) and call this unweighted – the result is identical.

Value

A `dist` object.

See Also

Other clustering: `TDA_CLUSTER`, `TDA_MDS`, `tda_conjoint()`, `tda_cutree()`, `tda_dma()`

Examples

```
set.seed(1)
d <- data.frame(x1 = rnorm(5), x2 = rnorm(5))
tda_pdatd(d)
tda_pdatd(d, measure = 2) # city-block, not the euclidean default
tda_pdatd(d, variables = "x1") # a subset of the variables
```

`tda_pdatr`

Print all case pairs

Description

TDA's `pdatr`: writes the kept variables for every ordered pair of cases.

Usage

```
tda_pdatr(data, keep = names(data), ...)
```

Arguments

<code>data</code>	data frame.
<code>keep</code>	variables to keep (all by default).
<code>...</code>	passed to <code>tda_run</code> .

Value

a data frame with one row per ordered pair of cases: the case numbers *i* and *j*, then the kept variables of case *i* (suffix *.i*) and of case *j* (suffix *.j*).

Examples

```
# every ordered pair of cases, with x for both members of the pair
tda_pdatr(data.frame(id = 1:3, x = c(10, 20, 30)), keep = "x")
```

tda_pl	<i>Add a command to a plot</i>
--------	--------------------------------

Description

tda_pl reaches any of TDA's thirty-seven plot commands. The named functions are the common ones and only spell out their arguments; anything without a named wrapper goes through tda_pl with the command name.

Usage

```
tda_pl(p, cmd, ..., rhs = NULL)
```

```
tda_pl_axes(
  p,
  sc = NULL,
  ic = NULL,
  lty = NULL,
  lw = NULL,
  fs = NULL,
  tl = NULL,
  fmt = NULL,
  ...
)
```

```
tda_pl_frame(p, lty = NULL, lw = NULL, gray = NULL, ...)
```

```
tda_pl_grid(p, at_x = NULL, at_y = NULL, lty = NULL, lw = NULL, ...)
```

```
tda_pl_lines(
  p,
  x,
  y,
  select = NULL,
  by = NULL,
  lty = NULL,
  band = NULL,
  rows = NULL,
```

```
    gray = NULL,  
    ...  
)  
  
TDA_LTY  
  
tda_pl_points(  
    p,  
    x,  
    y,  
    symbol = 1,  
    size = NULL,  
    lty = 0,  
    lw = NULL,  
    gray = NULL,  
    ...  
)  
  
tda_pl_smooth(p, x, y, ns = 2, lty = NULL, gray = NULL, ...)  
  
tda_pl_circle(  
    p,  
    at,  
    r = 1,  
    angles = NULL,  
    lty = NULL,  
    lw = NULL,  
    gray = NULL,  
    ...  
)  
  
tda_pl_text(  
    p,  
    label,  
    at,  
    fs = NULL,  
    rotate = NULL,  
    white = NULL,  
    symbol = NULL,  
    center = NULL,  
    ...  
)  
  
tda_pl_rect(p, x, y, lty = NULL, lw = NULL, gray = NULL, rotate = NULL, ...)  
  
tda_pl_polyline(p, x, y, lty = NULL, lw = NULL, gray = NULL, ...)  
  
tda_pl_function(  

```

```

    p,
    expr,
    range,
    step = NULL,
    lty = NULL,
    lw = NULL,
    gray = NULL,
    deriv = NULL,
    ...
)

tda_ps_file(x)

```

Arguments

<code>p</code>	a <code>tda_ps</code> session.
<code>cmd</code>	the TDA command name, e.g. <code>"plotd"</code> , <code>"plsurf3"</code> .
<code>...</code>	further options for the underlying TDA command, e.g. <code>lt=</code> (line type) or <code>lw=</code> (line width in mm) for <code>tda_pl_grid</code> . Each command's full option list is TDA's – look it up with <code>tda_help("plxa")</code> , <code>tda_help("plxgrid")</code> , and so on (the name after <code>"pl"</code> in each function above); this page only names the options with their own R argument.
<code>rhs</code>	the command's right-hand side, when it takes one.
<code>sc</code>	for <code>tda_pl_axes</code> , the distance between major tick marks (TDA's default: chosen automatically). For <code>tda_pl_labels</code> , the label's distance from the top of the plot, in mm (default 0). For <code>tda_pl_axes</code> specifically, this (and <code>ic/lty/ lw/fs/tl</code> below) can also be a pair, <code>c(x,y)</code> , to give the x and y axis their value instead of sharing one – <code>sc = c(1, 0.1)</code> for very different scales on the two axes, say. A single value (the usual case) is still used for both, unchanged.
<code>ic</code>	for <code>tda_pl_axes</code> , the number of minor sub-intervals between major tick marks.
<code>lty</code>	line type, either an R name – <code>"solid"</code> , <code>"dashed"</code> , <code>"dotted"</code> , <code>"dotdash"</code> , <code>"longdash"</code> , <code>"twodash"</code> – or a TDA number. With <code>by=</code> and no <code>lty</code> , the groups cycle through that list in order, as they would in R.
<code>lw</code>	line width in mm, for <code>tda_pl_grid</code> , <code>tda_pl_axes</code> , <code>tda_pl_frame</code> , and <code>tda_pl_circle</code> .
<code>fs</code>	font size in mm, for <code>tda_pl_axes</code> (axis labels) and <code>tda_pl_labels</code> (the label itself).
<code>tl</code>	for <code>tda_pl_axes</code> , the length of the tick marks, in mm.
<code>fmt</code>	for <code>tda_pl_axes</code> , the numeric format for tick labels (TDA's <code>fmt=</code> on <code>plxa/plya</code> , e.g. <code>"4.1"</code> for one decimal place in a field 4 characters wide).
<code>gray</code>	for <code>tda_pl_smooth</code> , grey scale value (TDA's <code>gs=</code>).
<code>at_x, at_y</code>	for <code>tda_pl_grid</code> , where to draw the vertical and horizontal grid lines; default to the axis's tick positions.
<code>x, y</code>	for <code>tda_pl_lines/tda_pl_points</code> , a column name in the session's data, or the numbers themselves – R's <code>lines(x = 1:10, y = 1:10)</code> . A vector the same length as the session's data is added as a new column; a different length gets its

	own, separate data matrix for just that one series (TDA's <code>clear;+nvar(dfile=...)</code> trick, restored afterward), useful for overlaying a fitted curve at a different resolution than the raw data it was fit to – see tda_npreg 's example. For everything else reachable through <code>tda_pl</code> , variables or coordinates, as names in the session's data or as numbers; for <code>tda_pl_rect</code> , two opposite corners.
<code>select</code>	draw only this level of <code>by</code> – TDA's <code>sel=</code> , a row selection, said in R (the same name every other wrapper with a <code>sel=</code> uses).
<code>by</code>	a grouping column in the plot's data. Without <code>select</code> , one line is drawn per level.
<code>band</code>	a pair of column names holding the upper and lower bound of a confidence band, shaded behind the line the way <code>examples/ehhnew/ehc8.cf</code> does it.
<code>rows</code>	an explicit row selection, for drawing two series of different lengths out of one frame.
<code>symbol</code>	for <code>tda_pl_text</code> , draw a marker symbol at this position instead of (or as well as) the text (TDA's <code>s=</code> on <code>pltext</code>).
<code>size</code>	symbol size.
<code>ns</code>	degree of smoothing.
<code>at</code>	where to draw it, as a pair of coordinates.
<code>r</code>	radius of a circle, in the plot's units.
<code>angles</code>	for <code>tda_pl_circle</code> , the start and end angle in degrees, counterclockwise, of a partial arc rather than a full circle (TDA's optional <code>alpha,beta</code> on <code>ploto</code> 's right-hand side, <code>r[,alpha,beta]</code>); omit for a full circle.
<code>label</code>	text to draw.
<code>rotate</code>	rotation in degrees, for <code>tda_pl_text</code> and <code>tda_pl_rect</code> (around their reference point) and <code>tda_pl_ellipse</code> (of its main axis).
<code>white</code>	for <code>tda_pl_text</code> , draw the label on a white background rather than transparently.
<code>center</code>	for <code>tda_pl_text</code> , centre the string on <code>at</code> rather than TDA's default of starting it there (TDA's <code>sc=1</code> on <code>pltext</code>).
<code>expr</code>	an expression in TDA's language.
<code>range, step</code>	for <code>tda_pl_function</code> , the range of the function's argument and the increment to evaluate it at – TDA's <code>rx = a(d)b</code> , built from these so the range does not need to be written in TDA's step notation by hand.
<code>deriv</code>	for <code>tda_pl_function</code> , plot the function's first (1) or second (2) derivative instead of the function itself (TDA's <code>plotf1/plotf2</code>); omit to plot the function as written.

Value

The session, with the command added.

See Also

Other plotting: [plot.tda_pl\(\)](#), [tda_check_ps\(\)](#), [tda_pl_arc\(\)](#), [tda_pl_axis\(\)](#), [tda_pl_graph\(\)](#), [tda_pl_hist](#), [tda_pl_panel\(\)](#), [tda_pl_regression\(\)](#), [tda_pl_scatter\(\)](#), [tda_plot_ps\(\)](#), [tda_ps\(\)](#), [tda_ps3\(\)](#), [tda_read_ps\(\)](#)

Examples

```

set.seed(1)
d <- data.frame(v = rnorm(60, 50, 10))
p <- tda_ps(d, xlim = c(0, 100), ylim = c(0, 0.05))
p <- tda_pl_axes(p)

# plotd (a density plot) has no named wrapper -- it needs an evaluation
# grid (x=) and exactly one variable on the right-hand side, which is not
# the R column name but the internal one tda_ps() assigned; look it up in
# the session's xlab/xname the same way the named wrappers do
vname <- p$xname[p$xlabel == "v"]
p <- tda_pl(p, "plotd", x = "0(5)100", rhs = vname)
p

# every named wrapper, on one plot, so each one is a working example
# rather than a bare signature -- x/y take a column name or a number
# everywhere, and every one of these actually draws something
d2 <- data.frame(x = 1:5, y = c(2, 4, 5, 8, 9))
p2 <- tda_ps(d2, xlim = c(0, 6), ylim = c(0, 12), width = 90, height = 70)
p2 <- tda_pl_axes(p2)           # tick marks and their numbers
p2 <- tda_pl_frame(p2)          # a box around the plotting area
p2 <- tda_pl_grid(p2, lw = 0.15) # TDA's default (lw = 0.05mm) is
                                # thin to the point of illegible on a
                                # modern screen; heavier here on purpose
p2 <- tda_pl_lines(p2, "x", "y") # the data, joined point to point
p2 <- tda_pl_points(p2, "x", "y", symbol = 1, size = 4) # a marker per point
p2 <- tda_pl_smooth(p2, "x", "y", ns = 4, lty = "dashed") # a smoothed fit
p2 <- tda_pl_circle(p2, at = c(3, 5), r = 0.3) # mark a location
p2 <- tda_pl_text(p2, "peak", at = c(4.3, 9)) # a text label
p2 <- tda_pl_rect(p2, x = c(4, 5), y = c(11, 11.8)) # a rectangle
p2 <- tda_pl_polyline(p2, c(0.2, 1, 0.2), c(11, 11, 11.8)) # a raw shape
# its row along the top, so it doesn't cross the data line below
p2 <- tda_pl_function(p2, "sin(x)+11", range = c(0, 6), step = 0.2)
tda_ps_file(p2) # the PostScript path, once the session has been run
p2

```

tda_pl_arc

Curved arcs

Description

plotk: draws an arc, optionally ending in an arrow, between each consecutive pair of points – a curved alternative to [tda_pl_lines](#) for showing a directed relationship without the line hiding what is under it. Fixed along the way: a genuine rendering bug where a shallow arc (TDA's technique for one – a small slice of a large-radius circle) was measured by the full circle's radius when sizing the plot, inflating the bounding box to fit an almost entirely off-screen circle and shrinking everything else into a corner; the fix (using the arc segment's actual extent) is general, not specific to this function.

Usage

```
tda_pl_arc(p, x, y, curvature = 0, arrow = NULL, lty = NULL, lw = NULL, ...)
```

Arguments

p	a tda_ps session.
x, y	coordinates the arcs connect, in order; at least two points.
curvature	how much the arc bows out, as a fraction of the distance between the two points; 0 is a straight line. TDA's own sc=.
arrow	size of an arrowhead at the end of each arc, as c(length, width) in mm; no arrowhead if not given.
lty, lw	line type and line width in mm for the arc.
...	further options for the command.

Value

The session, with the command added.

See Also

Other plotting: [plot.tda_ple\(\)](#), [tda_check_ps\(\)](#), [tda_pl\(\)](#), [tda_pl_axis\(\)](#), [tda_pl_graph\(\)](#), [tda_pl_hist](#), [tda_pl_panel\(\)](#), [tda_pl_regression\(\)](#), [tda_pl_scatter\(\)](#), [tda_plot_ps\(\)](#), [tda_ps\(\)](#), [tda_ps3\(\)](#), [tda_read_ps\(\)](#)

Examples

```
p <- tda_ps(xlim = c(0, 10), ylim = c(0, 10))
p <- tda_pl_frame(p)
p <- tda_pl_arc(p, c(2, 8), c(2, 8), curvature = 3, arrow = c(1.5, 1))
plot(p)
```

tda_pl_axis

Draw a single axis, optionally at an explicit position

Description

tda_pl_axes draws both axes together, at the plot's boundary. tda_pl_axis draws one axis at a time and, via at, can place it anywhere in the plot rather than only at the boundary – TDA's technique for a second x or y axis (e.g. a top axis in different units from the bottom one), which tda_pl_axes alone cannot reach.

Usage

```
tda_pl_axis(
  p,
  which = c("x", "y"),
  sc = NULL,
  ic = NULL,
  lty = NULL,
  lw = NULL,
  fs = NULL,
  tl = NULL,
  fmt = NULL,
  at = NULL,
  dir = NULL,
  ...
)
```

Arguments

p a [tda_ps](#) session.

which "x" or "y", which axis to draw.

sc, ic, lty, lw, fs, tl, fmt as in [tda_pl_axes](#), for this one axis.

at the axis line's endpoints, c(xa, ya, xb, yb) in data units, when it is not simply the plot's boundary (TDA's `plxa(...)` = xa,ya,xb,yb right-hand side). Omit for an ordinary axis at the plot boundary.

dir which side of the axis line the tick labels are written on (TDA's `dir=`); only meaningful together with `at`.

... further options for `plxa/plya`.

Value

The session, with the command added.

See Also

Other plotting: [plot.tda_ple\(\)](#), [tda_check_ps\(\)](#), [tda_pl\(\)](#), [tda_pl_arc\(\)](#), [tda_pl_graph\(\)](#), [tda_pl_hist](#), [tda_pl_panel\(\)](#), [tda_pl_regression\(\)](#), [tda_pl_scatter\(\)](#), [tda_plot_ps\(\)](#), [tda_ps\(\)](#), [tda_ps3\(\)](#), [tda_read_ps\(\)](#)

Examples

```
p <- tda_ps(xlim = c(-2, 2), ylim = c(0, 1))
p <- tda_pl_axis(p, "x", sc = 1, ic = 2)
p <- tda_pl_axis(p, "y", sc = 0.5, ic = 0)
# a second x axis, drawn at y = 1 instead of the plot's bottom edge
p <- tda_pl_axis(p, "x", sc = 1, ic = 5, dir = 1, at = c(-1, 1, 1, 1))
```

tda_pl_graph	<i>Draw a graph with explicit node positions</i>
--------------	--

Description

plg is TDA's graph drawing command, and a different thing from pltree: it takes the node coordinates and the edges and draws them, with control over node shape and shading, edge curvature and arrowheads. pltree by contrast computes a tree layout and draws edges only.

Usage

```
tda_pl_graph(p, nodes, edges = NULL, ...)
```

Arguments

p	a tda_ps session.
nodes	a data frame with the node positions. It needs x and y; id numbers the nodes, defaulting to the row number, grey and shape set gs= and gt=, label a text string instead of the node number (str=), size the node's display size in mm (rd=), and lty, lw, fontsize its line type, line width, and font size (lt=/lw=/fs=).
edges	a data frame of edges with from and to, holding node ids. curve sets rd=, the radius that bends an edge, arrow a pair of arrowhead dimensions, label the value written on the edge (ic=), lty its line type, which is how the two directions of a reciprocal pair are told apart, lw its line width, fontsize the edge label's font size (fs=), and loop, for a self-edge, which side of the node it loops from (dir=).
...	further options for plg, e.g. nmax= (the maximum number of nodes, rarely needed since TDA's default of 100 is usually enough).

Value

The session, with the command added.

See Also

Other plotting: [plot.tda_ple\(\)](#), [tda_check_ps\(\)](#), [tda_pl\(\)](#), [tda_pl_arc\(\)](#), [tda_pl_axis\(\)](#), [tda_pl_hist](#), [tda_pl_panel\(\)](#), [tda_pl_regression\(\)](#), [tda_pl_scatter\(\)](#), [tda_plot_ps\(\)](#), [tda_ps\(\)](#), [tda_ps3\(\)](#), [tda_read_ps\(\)](#)

Examples

```
nodes <- data.frame(x = c(1, 3, 1, 3), y = c(2, 2, 1, 1))
edges <- data.frame(from = c(1, 1), to = c(2, 3))
p <- tda_ps(xlim = c(0, 4), ylim = c(0, 3), width = 40, height = 30)
p <- tda_pl_graph(p, nodes, edges)
```

tda_pl_hist

*Further plot commands***Description**

Named wrappers for the plot commands that take arguments worth spelling out. Everything else is reachable with [tda_pl](#).

Usage

```
tda_pl_density(  
  p,  
  x,  
  at,  
  bandwidth = NULL,  
  kernel = c("uniform", "triangle", "quartic", "epanechnikov"),  
  scale = NULL,  
  clip = TRUE,  
  lty = NULL,  
  lw = NULL,  
  gray = NULL,  
  ...  
)  
  
tda_pl_histogram(  
  p,  
  x,  
  breaks,  
  weights = NULL,  
  scale = NULL,  
  closed = c("left", "right"),  
  vertical_lines = TRUE,  
  clip = TRUE,  
  lty = NULL,  
  lw = NULL,  
  gray = NULL,  
  ...  
)  
  
tda_pl_contour(  
  p,  
  expr,  
  levels,  
  resolution = NULL,  
  lty = NULL,  
  lw = NULL,  
  gray = NULL,
```

```

    ...
)

tda_pl_curve(
  p,
  x,
  y,
  symbol = NULL,
  size = NULL,
  lty = NULL,
  lw = NULL,
  gray = NULL,
  smooth = NULL,
  ...
)

tda_pl_ellipse(
  p,
  at,
  axes = c(1, 0.5),
  rotate = NULL,
  lty = NULL,
  lw = NULL,
  gray = NULL,
  ...
)

tda_pl_labels(
  p,
  label,
  which = c("title", "x", "y"),
  sc = NULL,
  fs = NULL,
  ...
)

tda_pl_grid_lines(p, ...)
```

Arguments

<code>p</code>	a <code>tda_ps</code> session.
<code>x, y</code>	variables in the session's data, or coordinates.
<code>at</code>	evaluation points for a density, or a pair of coordinates.
<code>bandwidth</code>	kernel bandwidth for <code>tda_pl_density</code> ; TDA's default is 1.
<code>kernel</code>	kernel shape for <code>tda_pl_density</code> : "uniform" (default), "triangle", "quartic", or "epanechnikov".
<code>scale</code>	for <code>tda_pl_histogram</code> or <code>tda_pl_density</code> , a scaling factor applied to the bar heights or curve (<code>plot</code> / <code>plotd</code> 's <code>dscal</code> -, default 1).

clip	for tda_pl_histogram or tda_pl_density, clip to the plot's declared range (the default); FALSE turns this off (ploth/plotd's nc=1).
lty, lw, gray	line type, line width in mm, and grey level from 0 (black) to 1 (white), for tda_pl_histogram, tda_pl_density, tda_pl_contour, tda_pl_curve, and tda_pl_ellipse.
...	further options for the underlying TDA command – look them up with tda_help(), e.g. tda_help("ploth") for tda_pl_histogram; see tda_pl .
breaks	interval boundaries for a histogram.
weights	for tda_pl_histogram, a variable in the session's data giving each case's weight.
closed	for tda_pl_histogram, which side of each interval is closed: "left" (default) or "right" – ploth's own s= (0/1).
vertical_lines	for tda_pl_histogram, draw the vertical lines between bars (the default); FALSE omits them (ploth's ns=1).
expr	an expression in TDA's language.
levels	contour levels for tda_pl_contour.
resolution	for tda_pl_contour, the evaluation grid size, as c(nx, ny) or one number for both; TDA's default is c(10, 10).
symbol	marker symbol, for tda_pl_curve.
size	marker size in mm, for tda_pl_curve.
smooth	the number of intervals for Akima smoothing of a convex hull's outline; TDA's ns=. ploth smooths only when this is 2 or more AND the hull has more than two vertices – below either it draws the plain outline (t_plot.c, pl_ploth).
axes	for tda_pl_ellipse, the half-lengths of the main and second axis, c(a, b), in the plot's x and y units.
rotate	rotation in degrees, for tda_pl_ellipse, of its main axis.
label	the text tda_pl_labels places at the top of the plot.
which	which label to set: "title" (TDA's plabel, at the top of the plot), "x" (pxlabel, below the x axis) or "y" (pylabel, left of the y axis). All three reach the same underlying TDA command (pl_label()); none of the three appear in TDA's help text, despite being real, working commands every shipped multi-axis example relies on), so they share every other option here too.
sc	for tda_pl_labels, the label's distance from the top of the plot, in mm (default 0).
fs	font size in mm, for tda_pl_labels.

Value

The session, with the command added.

See Also

Other plotting: [plot.tda_ple\(\)](#), [tda_check_ps\(\)](#), [tda_pl\(\)](#), [tda_pl_arc\(\)](#), [tda_pl_axis\(\)](#), [tda_pl_graph\(\)](#), [tda_pl_panel\(\)](#), [tda_pl_regression\(\)](#), [tda_pl_scatter\(\)](#), [tda_plot_ps\(\)](#), [tda_ps\(\)](#), [tda_ps3\(\)](#), [tda_read_ps\(\)](#)

Examples

```

set.seed(1)
d <- data.frame(v = round(rnorm(60) * 10 + 50, 1))
# ploth's bars are density-scaled (each bar's area, not its height, is
# its share of the cases), not raw counts -- ylim has to fit that
# small scale or the bars are real but too short to see
p <- tda_ps(d, xlim = c(0, 100), ylim = c(0, 0.05))
p <- tda_pl(p, "plxa", sc = 20, ic = 1)

# ploth takes the interval boundaries with breaks=, not tp=: passing tp=
# is silently ignored and every bar comes out flat.
tda_pl_histogram(p, "v", breaks = seq(0, 100, 10))

# plotd is a kernel density estimate, and at= are the points it is
# evaluated at rather than the intervals of a histogram.
tda_pl_density(p, "v", at = seq(0, 100, 5))

# a curve through data, a contour of an expression in two variables, an
# ellipse, a grid of light lines at the axis ticks, and a title
d2 <- data.frame(x = 1:5, y = c(2, 4, 5, 8, 9))
p2 <- tda_ps(d2, xlim = c(0, 6), ylim = c(0, 10))
p2 <- tda_pl_axes(p2)
p2 <- tda_pl_curve(p2, "x", "y")
p2 <- tda_pl_contour(p2, "x^2+y^2", levels = c(10, 30, 50))
p2 <- tda_pl_ellipse(p2, at = c(3, 5), axes = c(1.5, 0.8))
p2 <- tda_pl_grid(p2, lw = 0.15)
p2 <- tda_pl_labels(p2, "my plot title")
p2

```

tda_pl_panel

Start a new panel within the same session

Description

TDA can lay out more than one plot on a single page by calling `psetup` again, at a different physical position (`psorg=`) – this is that, reachable with the same `xlim/ylim/ width/height` vocabulary [tda_ps](#) itself uses to start the first one, rather than a raw, hand-built `psetup` block. Every `tda_pl_*` command after this one draws into the new panel; the commands already added stay in the first (or previous) panel.

Usage

```

tda_pl_panel(
  p,
  origin,
  width = NULL,
  height = NULL,
  xlim = NULL,
  ylim = NULL,

```

```
    ...
  )
```

Arguments

<code>p</code>	a <code>tda_ps</code> session.
<code>origin</code>	the new panel's position relative to the previous panel's own (the session's <code>origin=</code> for the first panel, TDA's default if that was not given), <code>c(dx, dy)</code> in mm – <code>c(width, 0)</code> of the previous panel places this one immediately to its right, for instance. Not an absolute page position: TDA's <code>psorg=</code> is itself relative to whatever the previous one was (PostScript's cumulative <code>translate</code>), and this mirrors that rather than asking the caller for TDA's undocumented default starting point. origin is the gap between each panel's <i>physical position</i> , not between their visible content: an axis tick's label extends a few mm past the plot's logical boundary (the width the tick mark itself sits at), so an <code>origin</code> exactly equal to the previous panel's width (no gap at all) can let the two panels' own tick labels collide right at the seam. Adding a real gap on top of <code>width – 10-15mm</code> is normally enough – keeps each panel's labels clear of its neighbour's.
<code>width, height</code>	size of the new panel, in mm (TDA's <code>pxlen=</code> / <code>pylen=</code>); default to the same size the session itself started with.
<code>xlim, ylim</code>	the new panel's coordinate range; default to the same range the session itself started with.
<code>...</code>	further <code>psetup</code> options.

Value

The session, with the new panel started.

See Also

Other plotting: `plot.tda_ple()`, `tda_check_ps()`, `tda_pl()`, `tda_pl_arc()`, `tda_pl_axis()`, `tda_pl_graph()`, `tda_pl_hist`, `tda_pl_regression()`, `tda_pl_scatter()`, `tda_plot_ps()`, `tda_ps()`, `tda_ps3()`, `tda_read_ps()`

Examples

```
p <- tda_ps(xlim = c(0, 6), ylim = c(-1, 1), width = 50, height = 40)
p <- tda_pl_axes(p, sc = 1)
p <- tda_pl_function(p, "sin(x1)", range = c(0, 6))
p <- tda_pl_text(p, "Plot 1", at = c(4, 0.2))
# 50 (the previous panel's width) plus a real gap, or its own
# tick labels collide with this new panel's own.
p <- tda_pl_panel(p, origin = c(65, 0), width = 50, height = 30)
p <- tda_pl_axes(p, sc = 1)
p <- tda_pl_function(p, "sin(x1)", range = c(0, 6))
p <- tda_pl_text(p, "Plot 2", at = c(4, 0.2))
```

tda_pl_regression	<i>Add a fitted regression curve or convex hulls to a plot</i>
-------------------	--

Description

tda_pl_regression draws the fitted curve that TDA's screen-only xreg command used to add to an X11 scatterplot – the X11 front end was never part of this source tree, so the three fits it offered are reproduced here and drawn through the ordinary PostScript session: a lowess curve, a least-squares line, or an L1-norm (least-absolute-deviations) line, one per group. The lowess curve is TDA's, drawn by scplot – the same one [tda_pl_scatter](#) draws, and the one xreg drew. (npreg(opt = 4) runs the same C routine but cannot give the same curve: it resets its band width to 1 when it is below epsilon, before the lowess branch, so the delta shortcut is never 0, while scplot passes 0.) The least-squares line is ordinary regression of y on x; the L1 line is TDA's l1reg via [tda_l1reg](#). Like xreg, the straight-line fits are drawn from the group's smallest x to its largest.

Usage

```
tda_pl_regression(
  p,
  x,
  y,
  type = c("lowess", "least_squares", "l1"),
  by = NULL,
  bandwidth = 0.5,
  lty = NULL,
  lw = NULL,
  gray = NULL,
  ...
)

tda_pl_hull(
  p,
  x,
  y,
  by = NULL,
  lty = NULL,
  lw = NULL,
  gray = NULL,
  symbol = NULL,
  size = NULL,
  smooth = NULL,
  expand = NULL,
  ...
)
```

Arguments

p A plot session started with [tda_ps](#).

x, y	Column names in the session's data.
type	The fit: "lowess" (the default, as in xreg), "least_squares", or "l1".
by	Optional name of a grouping column; one fit (or hull) per group. Without it, all points are one group.
bandwidth	The lowess span, xreg's sig= (default 0.5, clamped there to (0, 1)).
lty, lw, gray	Line type, width (mm), and gray level for the drawn curve.
...	Further options passed to the drawing command.
symbol, size	for tda_pl_hull, a marker symbol and its size in millimetres drawn at the points; plotch's s= and fs=.
smooth	the number of intervals for Akima smoothing of the hull's outline; plotch's ns=.
expand	a margin added to a smoothed hull, in millimetres; TDA's ic=. Each point of the smoothed outline is pushed away from the hull's centroid in proportion to its distance from it along each axis, so the outline grows without changing shape (t_plot.c, pl_plotch).

Details

tda_pl_hull is xconh: the convex hull of each group's points, drawn as a closed outline.

The third screen command, xplotf, plotted values straight from a file; [tda_pl_scatter](#) and [tda_pl_lines](#) with raw vectors already do that.

Value

The session, with the curve(s) added. The fitted coefficients (for the straight-line types) or curves are kept in p\$fits, one entry per call and group.

See Also

Other plotting: [plot.tda_ple\(\)](#), [tda_check_ps\(\)](#), [tda_pl\(\)](#), [tda_pl_arc\(\)](#), [tda_pl_axis\(\)](#), [tda_pl_graph\(\)](#), [tda_pl_hist](#), [tda_pl_panel\(\)](#), [tda_pl_scatter\(\)](#), [tda_plot_ps\(\)](#), [tda_ps\(\)](#), [tda_ps3\(\)](#), [tda_read_ps\(\)](#)

Examples

```
set.seed(1)
d <- data.frame(x = round(runif(40, 0, 10), 2))
d$y <- round(1 + 0.8 * d$x + rnorm(40), 2)
p <- tda_ps(d, xlim = c(0, 10), ylim = c(0, 12))
p <- tda_pl_frame(p)
p <- tda_pl_scatter(p, "x", "y", symbol = 1)
p <- tda_pl_regression(p, "x", "y", type = "least_squares",
                      lty = "dashed")
p$fits
```

tda_pl_scatter	<i>Scatterplots, sunflower plots, and lowess smoothing</i>
----------------	--

Description

scplot: three related ways of showing an x/y relationship, all in one command via `type` – plain symbols (the default), a sunflower plot (symbol density shown as petal count, for data too crowded for plain symbols to read), or a lowess-smoothed curve.

Usage

```
tda_pl_scatter(
  p,
  x,
  y,
  type = c("points", "sunflower", "lowess"),
  symbol = NULL,
  size = NULL,
  grid = NULL,
  bandwidth = NULL,
  select = NULL,
  clip = TRUE,
  lty = NULL,
  lw = NULL,
  ...
)
```

Arguments

<code>p</code>	a <code>tda_ps</code> session.
<code>x, y</code>	variables in the session's data, or coordinates.
<code>type</code>	"points" (default), "sunflower", or "lowess".
<code>symbol</code>	marker symbol, for <code>type = "points"</code> .
<code>size</code>	marker (or, for <code>type = "sunflower"</code> , petal) size in mm.
<code>grid</code>	for <code>type = "sunflower"</code> , the grid the plot area is divided into to group nearby points, as <code>c(nx, ny)</code> or one number for both; TDA's default is <code>c(1, 1)</code> .
<code>bandwidth</code>	for <code>type = "lowess"</code> , the smoothing factor; TDA's default is 0.5.
<code>select</code>	a case-selection expression, TDA's <code>sel=</code> .
<code>clip</code>	clip points to the plot's declared range (the default); FALSE turns this off (<code>scplot</code> 's <code>nc=1</code>), letting a point outside the range draw past the frame.
<code>lty, lw</code>	line type and width for <code>type = "lowess"</code> 's fitted curve.
<code>...</code>	further options for the command, e.g. <code>d=/ns=</code> for <code>type = "lowess"</code> – distinct from <code>bandwidth</code> 's <code>sig=</code> , and not promoted to their named parameters since TDA's help text for them ("optional parameter for <code>opt=3</code> ") does not say what they control.

Value

The session, with the command added.

See Also

Other plotting: [plot.tda_ple\(\)](#), [tda_check_ps\(\)](#), [tda_pl\(\)](#), [tda_pl_arc\(\)](#), [tda_pl_axis\(\)](#), [tda_pl_graph\(\)](#), [tda_pl_hist](#), [tda_pl_panel\(\)](#), [tda_pl_regression\(\)](#), [tda_plot_ps\(\)](#), [tda_ps\(\)](#), [tda_ps3\(\)](#), [tda_read_ps\(\)](#)

Examples

```
set.seed(1)
d <- data.frame(x = round(runif(30, 0, 10), 1),
                y = round(runif(30, 0, 10), 1))
p <- tda_ps(d, xlim = c(0, 10), ylim = c(0, 10))
p <- tda_pl_frame(p)
p <- tda_pl_scatter(p, "x", "y", symbol = 1)
plot(p)

# a lowess-smoothed curve instead, on data with a real nonlinear trend
d2 <- data.frame(x = sort(round(runif(30, 0, 10), 1)))
d2$y <- round(0.5 * d2$x^1.5 + rnorm(30, sd = 1), 2)
p2 <- tda_ps(d2, xlim = c(0, 10), ylim = c(0, 20))
p2 <- tda_pl_frame(p2)
p2 <- tda_pl_scatter(p2, "x", "y", type = "lowess", bandwidth = 0.6,
                    lty = "dashed")
plot(p2)
```

tda_plcirc3

Circle in a 3-d plot

Description

TDA's `plcirc3` on a `psetup3` system.

Usage

```
tda_plcirc3(radius, center = c(0, 0, 0), normal = c(0, 0, 1), ...)
```

Arguments

<code>radius</code>	circle radius.
<code>center</code>	xyz center.
<code>normal</code>	direction vector of the circle plane.
<code>...</code>	passed to tda_run .

Value

path of the PostScript file, invisibly.

Examples

```
# a unit circle in the xy plane, drawn on its own psetup3 system
f <- tda_plcirc3(1, center = c(0, 0, 0), normal = c(0, 0, 1))
tda_plot_ps(tda_read_ps(f))

# the same command added to a session, so it can share a plot with
# other 3-d commands: TDA's own option names, xyz= and dvec=
p <- tda_ps3(xlim = c(-2, 2), ylim = c(-2, 2), zlim = c(-2, 2))
p <- tda_pl(p, "plcirc3", xyz = "0,0,0", dvec = "0,0,1", rhs = 1)
p <- tda_pl(p, "plcirc3", xyz = "0,0,0", dvec = "0,1,0", rhs = 1)
plot(p)
```

tda_plglob3

Globe in a 3-d plot

Description

TDA's plglob3: longitude/latitude grid on a sphere.

Usage

```
tda_plglob3(radius, lon = seq(-150, 150, 30), lat = seq(-60, 60, 30), ...)
```

Arguments

radius	sphere radius.
lon, lat	grid sequences (degrees).
...	passed to tda_run .

Value

path of the PostScript file, invisibly.

Examples

```
# a globe with meridians every 30 degrees and parallels at -60..60
f <- tda_plglob3(1)
tda_plot_ps(tda_read_ps(f))

# a coarser grid, as one command of a session
p <- tda_ps3(xlim = c(-2, 2), ylim = c(-2, 2), zlim = c(-2, 2))
p <- tda_pl(p, "plglob3", lon = "-90,0,90", lat = "-45,0,45", rhs = 1)
plot(p)
```

tda_plot_ps

*Draw TDA's PostScript output with grid***Description**

Replays what [tda_read_ps](#) parsed onto the current grid device, so a plot TDA produced appears in an R graphics device rather than only in a .ps file.

Usage

```
tda_plot_ps(x, newpage = TRUE, cex = NULL, ...)
```

Arguments

x	a path, a TDA run, or the result of tda_read_ps .
newpage	start a new page before drawing.
cex	character expansion for the text, relative to the size TDA asked for. The sizes in the file are PostScript points against a plot of perhaps 80mm; drawn into a device several times that, they come out small, so the default scales them with the drawing, times <code>getOption("tdaR.ps.cex", 1)</code> .
...	passed to tda_read_ps .

Value

The parsed operations, invisibly.

See Also

Other plotting: [plot.tda_ple\(\)](#), [tda_check_ps\(\)](#), [tda_pl\(\)](#), [tda_pl_arc\(\)](#), [tda_pl_axis\(\)](#), [tda_pl_graph\(\)](#), [tda_pl_hist](#), [tda_pl_panel\(\)](#), [tda_pl_regression\(\)](#), [tda_pl_scatter\(\)](#), [tda_ps\(\)](#), [tda_ps3\(\)](#), [tda_read_ps\(\)](#)

Examples

```
# a PostScript file with something in it: five points on a quadratic.
# psetup needs its ranges -- axes alone draw nothing, and tda_plot_ps
# then has nothing to render.
r <- tda_run(c("nvar(noc = 5, X = case, Y = case * case);",
              "psfile = p.ps;",
              "psetup(pxlen = 90, pylen = 50, pxa = 0,6, pya = 0,30);",
              "plxa(sc = 1); plya(sc = 10); plframe;",
              "plot = X,Y;", "psclose;"))
p <- tda_plot_ps(file.path(r$dir, "p.ps"))
```

tda_plotcm	<i>Contour plot of a matrix</i>
------------	---------------------------------

Description

TDA's plotcm over a value matrix at the given levels.

Usage

```
tda_plotcm(z, levels, ...)
```

Arguments

z	value matrix.
levels	contour levels.
...	passed to tda_run .

Value

path of the PostScript file, invisibly.

Examples

```
# a bowl-shaped surface over an 8 x 8 grid: contours at three heights
z <- outer(1:8, 1:8, function(i, j) (i - 4)^2 + (j - 4)^2)
f <- tda_plotcm(z, levels = c(2, 6, 12))
tda_plot_ps(tda_read_ps(f))
```

tda_plotr	<i>Grey-scale relief of a matrix</i>
-----------	--------------------------------------

Description

TDA's plotr.

Usage

```
tda_plotr(z, grey_range = c(0, 1), ...)
```

Arguments

z	value matrix.
grey_range	two greys mapped to the value extremes.
...	passed to tda_run .

Value

path of the PostScript file, invisibly.

Examples

```
# cells shaded from white (smallest value) to black (largest)
f <- tda_plotr(outer(1:5, 1:5, "+"))
tda_plot_ps(tda_read_ps(f))
```

tda_plotsp

Akima-smoothed curve through points

Description

TDA's plotsp.

Usage

```
tda_plotsp(x, y, ...)
```

Arguments

`x, y` coordinates.
`...` passed to `tda_run`.

Value

path of the PostScript file, invisibly.

Examples

```
# a smooth curve through five points that zigzag between 0 and 1
f <- tda_plotsp(x = c(0, 1, 2, 3, 4), y = c(0, 1, 0, 1, 0))
tda_plot_ps(tda_read_ps(f))
```

tda_plsurf3	<i>Parametric surface in a 3-d plot</i>
-------------	---

Description

TDA's plsurf3: x, y, z as expressions in u and v.

Usage

```
tda_plsurf3(  
  fx,  
  fy,  
  fz,  
  u_range = c(-1, 1, 5, 10),  
  v_range = c(-1, 1, 5, 10),  
  ...  
)
```

Arguments

fx, fy, fz coordinate expressions in u and v (TDA syntax).
u_range, v_range c(from, to, grid_lines, points_per_line).
... passed to [tda_run](#).

Value

path of the PostScript file, invisibly.

Examples

```
# z = sin(u) cos(v) over a 5 x 5 grid of lines, 12 points each  
f <- tda_plsurf3(fx = "u", fy = "v", fz = "sin(u) * cos(v)",  
               u_range = c(-2, 2, 5, 12),  
               v_range = c(-2, 2, 5, 12))  
tda_plot_ps(tda_read_ps(f))  
# tda_pl_surface3() adds the same surface to a tda_ps3() session
```

tda_polygons	<i>Build polygons from line segments</i>
--------------	--

Description

sdcpol takes a set of line segments, finds where they meet and assembles the polygons they enclose. It works on a plain table of segments rather than on a [tda_spatial](#) structure, and with option = 3 it writes a spatial data file the rest of the family can read.

Usage

```
tda_polygons(segments, option = 3, dir = tempfile("tda"), ...)
```

Arguments

segments	a data frame of four columns: the two ends of each segment, as x1, y1, x2, y2.
option	1 prints the node list of the graph the segments form, 2 its edge list, 3 constructs the polygons.
dir	working directory for the run.
...	further options for sdcpol; tol is the tolerance for treating two points as the same, default 1e-4.

Value

For option = 3 a [tda_spatial](#) structure; otherwise the run, with the table in \$table.

See Also

Other spatial analysis: [plot.tda_spatial\(\)](#), [tda_map\(\)](#), [tda_read_dbf\(\)](#), [tda_read_shapefile\(\)](#), [tda_read_spatial](#), [tda_sd\(\)](#), [tda_sd_analyses](#), [tda_sd_ps3\(\)](#), [tda_spatial\(\)](#)

Examples

```
sq <- data.frame(x1 = c(0, 1, 1, 0), y1 = c(0, 0, 1, 1),
                 x2 = c(1, 1, 0, 0), y2 = c(0, 1, 1, 0))
tda_polygons(sq)
```

tda_ps

*A TDA plot***Description**

Opens a plot session. Drawing commands are added to it and it is run when the plot is drawn or its file is asked for, which mirrors how TDA works: a PostScript file is opened, a coordinate system set up, commands issued, and the file closed.

Usage

```
tda_ps(
  data = NULL,
  width = 100,
  height = 70,
  xlim = NULL,
  ylim = NULL,
  file = "plot.ps",
  origin = NULL,
  log = NULL,
  ...
)
```

Arguments

<code>data</code>	optional data frame the drawing commands may refer to.
<code>width, height</code>	size of the plotting area in millimetres.
<code>xlim, ylim</code>	ranges of the coordinate system.
<code>file</code>	name for the PostScript file.
<code>origin</code>	the session's physical position on the page, $c(x, y)$ in mm from the page origin (TDA's <code>psorg=</code>). Only useful together with tda_pl_panel , since it fixes a known starting point that panel's <code>origin=</code> (relative to the previous panel) is added onto; omit it for an ordinary, single-panel session, where TDA's default page position is used.
<code>log</code>	which axes are logarithmic (TDA's <code>pxa(log=1)/pya(log=1)</code> , a per-axis option nested inside <code>pxa=/pya=</code> rather than an ordinary <code>psetup</code> option): any of "x", "y", "xy", or ""/NULL (the default) for neither.
<code>...</code>	further options for <code>psetup</code> .

Value

A `tda_ps` session.

See Also

Other plotting: [plot.tda_ple\(\)](#), [tda_check_ps\(\)](#), [tda_pl\(\)](#), [tda_pl_arc\(\)](#), [tda_pl_axis\(\)](#), [tda_pl_graph\(\)](#), [tda_pl_hist](#), [tda_pl_panel\(\)](#), [tda_pl_regression\(\)](#), [tda_pl_scatter\(\)](#), [tda_plot_ps\(\)](#), [tda_ps3\(\)](#), [tda_read_ps\(\)](#)

Examples

```
d <- data.frame(x = 1:5, y = c(2, 4, 5, 8, 9))
p <- tda_ps(d, xlim = c(0, 6), ylim = c(0, 10))
p <- tda_pl_axes(p)
p <- tda_pl_lines(p, "x", "y")
p
```

tda_ps_objects

What is inside a TDA PostScript file

Description

Reopens a PostScript file that TDA wrote (xopen) and lists the plot objects in it (xlog1). TDA numbers each thing it draws – an axis, a series, a convex hull – and records the command that drew it as a %\#N: command comment. This reports that list, which is how you find the number to pass to a later xdelete.

Usage

```
tda_ps_objects(file, options = list(), dir = tempfile("tda"), ...)
```

Arguments

file	a PostScript file written by TDA.
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to tda_run .

Details

The coordinate system and bounding box come back as attributes, because they are what decide whether a second plot can be added to this file at all.

Value

A data frame with object, the number, and command, the command that drew it – zero rows if the file holds none. `attr(x, "xlim")`, `attr(x, "ylim")` and `attr(x, "bbox")` carry the coordinate system; `attr(x, "run")` carries the run.

See Also

[tda_read_ps](#), [tda_combine_ps](#)

Examples

```
d <- data.frame(X = 1:10, Y = c(2, 4, 3, 6, 5, 8, 7, 10, 9, 12))
p <- tda_ps(d, file = "p.ps", xlim = c(0, 11), ylim = c(0, 13))
tda_ps_objects(tda_ps_file(p))
```

tda_ps3

*Three-dimensional plots***Description**

tda_ps3 opens a plot session using psetup3, TDA's three-dimensional coordinate setup, and the rest add to it: points, text, a polyline and a surface, each taking three coordinates where their two-dimensional counterparts take two. Note that the 3-D commands are the only ones that emit rotate and scale, which [tda_read_ps](#) does not yet interpret, so the replayed picture may be positioned wrongly even where the PostScript itself is correct.

Usage

```
tda_ps3(
  data = NULL,
  width = 100,
  xlim = c(0, 100),
  ylim = c(0, 100),
  zlim = c(0, 100),
  file = "plot.ps",
  ...
)
```

```
tda_pl_points3(
  p,
  x,
  y,
  z,
  symbol = NULL,
  size = NULL,
  lty = 0,
  lw = NULL,
  gray = NULL,
  arrow = NULL,
  ...
)
```

```
tda_pl_text3(
  p,
  label,
  at,
  fs = NULL,
```

```

    rotate = NULL,
    center = NULL,
    symbol = NULL,
    white = NULL,
    ...
)

tda_pl_surface3(
  p,
  ru,
  rv,
  f1,
  f2,
  f3,
  contour = NULL,
  lty = NULL,
  lw = NULL,
  gray = NULL,
  ...
)

tda_pl_curve3(p, rx, f1, f2, f3, lty = NULL, lw = NULL, gray = NULL, ...)

tda_pl_lines3(
  p,
  x,
  y,
  z,
  symbol = NULL,
  size = NULL,
  lty = NULL,
  lw = NULL,
  gray = NULL,
  arrow = NULL,
  ...
)

```

Arguments

<code>data</code>	a data frame, or nothing. Coordinates can also be handed to the drawing commands as plain vectors, which are folded into the session's data as they arrive.
<code>width</code>	size of the plotting area in millimetres.
<code>xlim, ylim, zlim</code>	ranges of the three axes.
<code>file</code>	name for the PostScript file.
<code>...</code>	further options for the command.
<code>p</code>	a <code>tda_ps</code> session opened by <code>tda_ps3</code> .

x, y, z	coordinates, either as numbers or as names of columns in the session's data. plot3 beneath tda_pl_lines3 names three variables rather than taking coordinates, so loose vectors are folded into the session's data for it; either form works. At least two points are needed in both, since these draw a polyline and one triple is not a line.
symbol, size, lty, lw, gray, arrow	for tda_pl_points3 and tda_pl_lines3: marker symbol and size, line type, line width, grey level, and an arrowhead at the end, as c(length, width) in mm – plot3/plot3's s=/fs=/ lt=/lw=/gs=/a=. tda_pl_text3 also takes symbol, drawn at the same position as the text.
label	text to draw.
at	where to draw it, as a triple of coordinates.
fs	font size in mm, for tda_pl_text3.
rotate	rotation in degrees, for tda_pl_text3.
center	for tda_pl_text3, centre the string on at rather than starting it there; pltext3's sc=.
white	for tda_pl_text3, draw the label on a white background rather than transparently; pltext3's wf=1.
ru, rv	for tda_pl_surface3, the two parameters' ranges and grid resolution, each as "a,b,n,m": from a to b, evaluated at n points, drawn with m grid lines. Keep n modest – 10 or so – if gray is also given: TDA shades every individual grid cell of the mesh, and the PostScript output size (and the time to parse and render it) grows with n squared, not linearly. Checked: an otherwise ordinary n = 30 surface produced over a million lines of PostScript once gray was added, and took long enough to parse that it looked hung rather than merely slow; n = 5 renders in about a second.
f1, f2, f3	the x, y and z coordinate as expressions in the curve's or surface's parameter(s) (x for tda_pl_curve3; u and v for tda_pl_surface3) – parametric, not a single $z = f(x,y)$: for a plain surface height field, use f1 = "u", f2 = "v", and the height expression itself for f3.
contour	for tda_pl_surface3, draw internal grid lines over the surface (cont = 1) in addition to its outline.
rx	for tda_pl_curve3, the range of its one parameter, as TDA's step notation "a(step)b" – not the comma form ru/rv take.

Value

A tda_ps session, with the command added.

See Also

Other plotting: [plot.tda_ple\(\)](#), [tda_check_ps\(\)](#), [tda_pl\(\)](#), [tda_pl_arc\(\)](#), [tda_pl_axis\(\)](#), [tda_pl_graph\(\)](#), [tda_pl_hist](#), [tda_pl_panel\(\)](#), [tda_pl_regression\(\)](#), [tda_pl_scatter\(\)](#), [tda_plot_ps\(\)](#), [tda_ps\(\)](#), [tda_read_ps\(\)](#)

Examples

```

x <- c(1, 3, 5, 7)
y <- c(2, 6, 3, 8)
z <- c(1, 4, 2, 6)
# no data frame needed: the vectors are folded in as they are used
p <- tda_ps3(xlim = c(0, 8), ylim = c(0, 10), zlim = c(0, 8))
p <- tda_pl_points3(p, x, y, z, symbol = 4)
p <- tda_pl_lines3(p, x, y, z)
p <- tda_pl_text3(p, "a corner", at = c(1, 2, 1))

# column names work too, resolved against the session's data
d <- data.frame(x = x, y = y, z = z)
p2 <- tda_ps3(d, xlim = c(0, 8), ylim = c(0, 10), zlim = c(0, 8))
p2 <- tda_pl_points3(p2, "x", "y", "z", symbol = 4)

# a surface, defined parametrically over a u,v grid rather than from
# data -- plsrf3 needs all three coordinate functions and both
# ranges explicitly, not a single z = f(x,y) expression
p3 <- tda_ps3(xlim = c(0, 8), ylim = c(0, 10), zlim = c(0, 8))
p3 <- tda_pl_surface3(p3, ru = "0,8,30,10", rv = "0,10,30,10",
                     f1 = "u", f2 = "v", f3 = "u*v/8")

# shaded, with internal contour lines -- n kept small (5, not 30) since
# gray= shades every grid cell individually; see the ru/rv argument
p3b <- tda_ps3(xlim = c(0, 8), ylim = c(0, 10), zlim = c(0, 8))
p3b <- tda_pl_surface3(p3b, ru = "0,8,5,5", rv = "0,10,5,5",
                     f1 = "u", f2 = "v", f3 = "u*v/8",
                     gray = 0.9, contour = TRUE)

# a parametric curve -- rx takes TDA's step notation (a(step)b),
# not the comma form the surface's ru/rv take
p4 <- tda_ps3(xlim = c(-2, 2), ylim = c(-2, 2), zlim = c(-2, 2))
p4 <- tda_pl_curve3(p4, rx = "0(0.1)6.28",
                  f1 = "cos(x)", f2 = "sin(x)", f3 = "x/3-1",
                  lty = "dotted", lw = 0.6)

# a single point: plotp3 (what tda_pl_points3 calls) rejects one triple
# outright, so this duplicates it -- the same marker drawn twice at the
# same place, joined by an invisible zero-length line, looks identical
# to drawing it once
p4 <- tda_ps3(xlim = c(0, 8), ylim = c(0, 10), zlim = c(0, 8))
p4 <- tda_pl_points3(p4, 4, 5, 3, symbol = 5, size = 3)

# a rotated, centred label on a white background, next to a bounding
# line -- a lone label with nothing else in the plot has no coordinate
# range to size the plot from and would come out unreadably small
p5 <- tda_ps3(xlim = c(0, 8), ylim = c(0, 10), zlim = c(0, 8))
p5 <- tda_pl_points3(p5, c(0, 8), c(0, 10), c(0, 8), symbol = 1,
                  size = 0.1)
p5 <- tda_pl_text3(p5, "hello", at = c(4, 5, 4), fs = 4, rotate = 30,
                  center = TRUE, white = TRUE)

```

tda_ptree	<i>Print a tree's data structure</i>
-----------	--------------------------------------

Description

ptree: the same tree [plot.tda_graph](#) draws (via `pltree`), but as data rather than a picture – one row per node, its parent, its first child, and its next sibling, the linked-list representation TDA itself builds internally to draw one.

Usage

```
tda_ptree(g, root = NULL, options = list(), dir = tempfile("tda"), ...)
```

Arguments

<code>g</code>	a tda_graph , undirected and shaped like a tree.
<code>root</code>	which node to treat as the root; the first node in <code>g</code> 's data by default.
<code>options</code>	a named list of further TDA options, passed through.
<code>dir</code>	working directory.
<code>...</code>	passed to tda_run .

Details

TDA's `gn=` (which graph, when more than one is open) is not optional here in practice: `ptree`, unlike `pltree`, never calls the setup step that gives it a safe default, so leaving it out reliably fails with “current graph is not a tree” even for a graph that unambiguously is one (`pltree` accepts the identical graph) – `gn = 1` is always sent explicitly to work around it.

Value

A data frame: `node`, `edge_from`, `edge_to`, `value`, `parent`, `first_child`, `next_sibling` (0 where a node has none).

See Also

Other graph analysis: [plot.tda_graph\(\)](#), [tda_dmet\(\)](#), [tda_g\(\)](#), [tda_g_analyses](#), [tda_graph\(\)](#), [tda_locate_line\(\)](#), [tda_ucl\(\)](#)

Examples

```
e <- data.frame(from = c(1, 1, 2), to = c(2, 3, 4), value = 1)
g <- tda_graph(e, directed = FALSE) # 4 nodes, 3 edges: a tree
tda_ptree(g, root = 1)
```

TDA_QRMODELS	<i>Qualitative response models</i>
--------------	------------------------------------

Description

Binary, ordinal and multinomial logit and probit models, conditional logit and simultaneous probit.

Usage

TDA_QRMODELS

```
tda_qreg(
  formula,
  data,
  model = "logit",
  nq = NULL,
  waves = NULL,
  min_waves = NULL,
  intercept = NULL,
  control = NULL,
  nintegral = NULL,
  tol_integral = NULL,
  parameterization = NULL,
  standardized = FALSE,
  predictions = FALSE,
  constraints = NULL,
  start = NULL,
  weights = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

formula	a two-sided formula.
data	a data frame.
model	a name from TDA_QRMODELS or a TDA model number.
nq	number of categories, for models 5 and 6. Filled in from the data if not given.
waves, min_waves	the number of waves and, of those, the minimum with valid data required per case (nw=/pmin=); only meaningful for models 7 and 8 (see the Panel models section).
intercept	whether to include an intercept. Defaults to whatever the formula itself says ($y \sim 0 + x$ or $y \sim x - 1$ already mean no intercept, the ordinary R way, and are honoured); set explicitly to override that.

control	convergence settings from tda_control .
nintegral, tol_integral	for models needing numerical integration (6-8), the integration control and its accuracy (qreg's <code>nhp=</code> / <code>eps=</code>).
parameterization	for models 6-8, how the between-wave correlation matrix is parameterised during estimation – 1 (default) direct; 2 a logistic-style transform keeping each correlation in $[-1, 1]$ by construction; 3 a Cholesky-style factorisation. A numerical-stability choice for the optimiser, not something that changes what the fitted correlations mean; qreg's <code>opt=</code> .
standardized	for a binary logit or probit model (<code>model = "logit"/"probit"</code>) fit to cross-sectional data (no waves), also compute standardized coefficients – qreg's <code>res=1</code> . A separate table, in <code>\$standardized</code> : each coefficient in standard-deviation units (<code>Coeff</code>), its exponential (<code>Exp(C)</code>), the exponential of the coefficient times the predictor's standard deviation (<code>Exp(C*SD)</code>), and that standard deviation itself (<code>Std.Dev.</code>). Only works for these two, cross-sectional models (TDA silently skips <code>res=</code> for any other), so requesting it for a different model errors here instead.
predictions	also write out each case's fitted probabilities – qreg's <code>df=</code> / <code>dt da=</code> . A separate table, in <code>\$predictions</code> : case and wave identifiers, the response, predictors, weight (if given), and <code>PROB/PROB0</code> , <code>PROB1</code> , ... (one <code>PROBn</code> per outcome category the model has). Unlike <code>standardized</code> , model-agnostic: works for every model, each getting whichever <code>PROB*</code> columns it actually computes.
constraints	for model 6 (<code>multivariate_probit</code>) or 8 (<code>simultaneous_probit</code>), linear constraints on the model's parameters, in TDA's <code>bN</code> numbering – qreg's <code>con=</code> , the same linear-constraint machinery tda_lsreg 's equality/inequality use, just addressed by number rather than by name. <code>bN</code> is the same numbering <code>tda_estimates()</code> 's <code>Idx</code> column already shows, so a first run without constraints tells you which number is which parameter. These two models introduce a correlation parameter for every pair of outcome categories ($\binom{nq}{2}$ of them) on top of the regular regression coefficients, and are usually only identified with most of them fixed – <code>examples/exam/qr6.cf</code> 's worked example fixes all ten (<code>nq = 5</code>) to zero, <code>constraints = paste0("b", 9:18, " = 0")</code> here, reproduced exactly. model's formula and variable list stay exactly the shape every other model already uses: a single response, one row per case.
start	optional starting values, in TDA's parameter order (the same order <code>bN</code> in <code>constraints</code> refers to, and <code>tda_estimates()</code> 's <code>Idx</code>) – qreg's <code>xp=</code> . Often needed in practice for models 6/8: TDA's automatic starting-value generator is not always good enough for these two models' extra correlation parameters, and a fit that converges cleanly with good starting values can fail outright without them.
weights	optional case weights, a column name in data or a vector as long as the data – TDA's <code>cwt = W;</code> , a separate, standalone command (not a per-command option), the same convention <code>tda_ple</code> / <code>tda_ltb</code> use.
options	a named list of further TDA options, passed through: the output files.
dir	working directory.
...	passed to tda_run .

Value

An object of class `tda_fit`. For the multivariate and simultaneous probits (models 6 and 8) it carries correlation, the estimated correlation matrix among the latent equations, which TDA computes only when asked for a parameter file. It additionally carries categories, the (weighted) count and percentage of each response category as `qreg` itself tabulates them before estimating. Cases with missing values are dropped before the fit, with a message (see `?tdaR`).

Models

`model` takes a name from `TDA_QRMODELS`, matched partially, or a TDA model number:

1	<code>logit</code>	binary logit
2	<code>probit</code>	binary probit
3	<code>ordinal_logit</code>	needs three or more categories
4	<code>ordinal_probit</code>	needs three or more categories
5	<code>multinomial_logit</code>	needs nq
6	<code>multivariate_probit</code>	needs nq
7	<code>conditional_logit</code>	panel: needs waves (2+) and <code>cbind()</code> terms – see Panel models
8	<code>simultaneous_probit</code>	panel: needs waves (2+) and <code>cbind()</code> terms – see Panel models

Panel models (waves)

Models 7 and 8 are TDA's panel models, taking each variable once per wave in a horizontal layout. Write each variable's per-wave columns as `cbind()`: `cbind(y1, y2) ~ cbind(x1, x2)` with waves = 2. A term with a single column is a time-constant predictor and is repeated across waves for you. Both models are fit without an intercept option (TDA's design; model 8 estimates one intercept per wave anyway, printed against Term = "W <k>", with the between-wave correlations as Sigma rows).

Missing data follows TDA's unbalanced-panel rule: a case whose *response* is missing in a wave simply has that wave dropped from its likelihood contribution (`min_waves` – TDA's `pmin=` – then sets how many valid waves a case needs to be used at all), while a missing *predictor* value has no such rule and drops the case, with a message.

`conditional_logit` is Chamberlain's fixed-effects logit; on two-wave data it agrees with `survival::clogit` stratified by case to full printed precision (pinned in the tests). `simultaneous_probit` fits one binary probit per wave with free cross-wave error correlations – for two waves, a bivariate probit. TDA's help text for this is incomplete: it names only model 5 as needing nq, though models 5 and 6 both require it, and it lists only the first five models, though eight are implemented – the table above is the full set. The binary models agree with `glm`: a binary logit is a binomial `glm` with a logit link, a binary probit one with a probit link.

See Also

Other regression: `TDA_FAMILIES`, `tda_freg()`, `tda_gdf()`, `tda_l1reg()`, `tda_lsreg()`, `tda_mlrc_design()`, `tda_mreg()`, `tda_nlreg()`, `tda_npreg()`, `tda_zreg()`

Examples

```
set.seed(23)
d <- data.frame(x = round(rnorm(200), 3))
```

```
d$y <- rbinom(200, 1, plogis(-0.5 + 1.4 * d$x))
qf <- tda_qreg(y ~ x, d, model = "logit")
coef(qf)
coef(glm(y ~ x, d, family = binomial())) # matches: a binary logit is a
                                           # binomial glm with a logit link
head(predict(qf))                        # fitted values, no newdata needed
```

tda_quant

Quantiles, histograms and one-way frequency tables

Description

tda_quant computes quantiles, tda_atab aggregates a variable into classes, which is how TDA makes a histogram, and tda_freq1 gives a one-way frequency table. All three describe the distribution of one variable at a time and return a table in the shape ... came in, one row per variable or class – unlike [tda_freq2](#) (a cross-tabulation of two variables, a matrix rather than a table) or [tda_loglin](#) (a many-way contingency table for model fitting), which need their pages.

Usage

```
tda_quant(..., options = list(), dir = tempfile("tda"))

tda_atab(
  ...,
  breaks,
  open = c("right", "left"),
  nonempty_only = FALSE,
  options = list(),
  dir = tempfile("tda")
)
```

Arguments

...	a data frame, a matrix, or vectors.
options	a named list of further TDA options, passed through.
dir	working directory.
breaks	for tda_atab, the class boundaries.
open	for tda_atab, which side of each class is left open: "right" (default) or "left" – atab's s= (0/1).
nonempty_only	for tda_atab, print only classes that contain at least one case (atab's r=1).

Details

tda_quant reports a fixed set of orders – 0.1 through 0.9 in steps of 0.1, plus the quartiles 0.25 and 0.75 – interpolated on the sorted values at position $p(n + 1)$ (the manual, section 6.2.3, gives the exact rule). That is the same definition as R's `quantile(x, type = 6)`, and the two agree to the last digit; `stats::quantile`'s default (`type = 7`) is a different interpolation and will not match.

Value

An object carrying a table: for `tda_quant`, one row per variable with the eleven quantiles `p10..p90`; for `tda_atab`, one row per class with its boundaries, count and percentage; for `tda_freq1`, one row per distinct value with count, percent and their cumulated versions.

See Also

Other descriptive statistics: [tda_brr\(\)](#), [tda_cov\(\)](#), [tda_dstat\(\)](#), [tda_freq\(\)](#), [tda_independence\(\)](#), [tda_ineq\(\)](#), [tda_loglin\(\)](#), [tda_rcorr\(\)](#), [tda_segr\(\)](#), [tda_subm\(\)](#)

Examples

```
set.seed(1)
d <- data.frame(x = sample(1:5, 60, replace = TRUE,
                          prob = c(0.1, 0.2, 0.4, 0.2, 0.1)))

tda_quant(d)
tda_atab(d, breaks = seq(0, 5, 1))
tda_freq1(d)
```

tda_rank_orders	<i>All rank orders of a given size</i>
-----------------	--

Description

`tda_rank_orders` enumerates every rank order of size objects, including all ways of tying them. For three objects there are 13: six strict orders, six with one pair tied, and one with all three tied.

Usage

```
tda_rank_orders(size = 2, options = list(), dir = tempfile("tda"), ...)
```

Arguments

- `size` number of objects to rank, 2 or more.
- `options` a named list of further TDA options, passed through. `m=` selects the enumeration variant.
- `dir` working directory.
- `...` passed to [tda_run](#).

Details

Useful as the sample space for a rank-order model – the set of outcomes a distribution over rankings has to sum over – which is otherwise fiddly to generate because of the ties.

Value

A data frame with one row per rank order: `ties`, the number of tie groups, then one column per object (`r1`, `r2`, ...) giving its rank. `attr(x, "run")` carries the run.

Examples

```
r <- tda_rank_orders(3)
nrow(r)      # 13
head(r)
table(r$ties) # 1 all tied, 6 one pair tied, 6 strict
```

tda_rap	<i>Rates in age-period form</i>
---------	---------------------------------

Description

TDA's rap: builds age-period tables from starting time, cohort, ending time and destination variables.

Usage

```
tda_rap(cohort, entry_year, exit_year, destination, years, ages, ...)
```

Arguments

- cohort, entry_year, exit_year, destination
integer vectors, one entry per case: birth year, year of entry into observation, year of exit, and the destination code (0 censored). The engine requires cohort <= entry_year <= exit_year.
- years, ages
length-2 table ranges.
- ...
passed to [tda_run](#).

Value

table, the rate table TDA wrote, and output.

Examples

```
# three subjects born 1960/1961, observed 1980-1986: the risk
# table fills along the Lexis diagonals
r <- tda_rap(cohort = c(60, 61, 60), entry_year = c(80, 82, 83),
            exit_year = c(85, 86, 86), destination = c(1, 1, 0),
            years = c(80, 86), ages = c(18, 26))
r$risk
```

tda_rates	<i>The estimated rate as a single table</i>
-----------	---

Description

The companion to [tda_survivor](#): the rate a model implies, with the same column names whichever model produced it. prate = at fit time is what asks TDA to compute it.

Usage

```
tda_rates(x)
```

Arguments

x a fit from [tda_rate](#) or [tda_coxph](#).

Value

A data frame with group, time, rate, survivor and density, or NULL if the fit carries no rate table. For a Cox fit rate is the baseline rate at each event time and cumrate the cumulative baseline rate beside it (already cumulative: TDA's CumRate column).

See Also

Other rate models: [tda_constrain\(\)](#), [tda_control\(\)](#), [tda_dple\(\)](#), [tda_split\(\)](#), [tda_survivor\(\)](#), [tda_transitions\(\)](#), [vcov.tda_fit\(\)](#)

Examples

```
d <- tda_rrdat()
f <- tda_rate(Surv(TFP, DES) ~ 1, d, prate = "0(10)300")
head(tda_rates(f))
```

tda_rcorr	<i>Rank correlation matrix</i>
-----------	--------------------------------

Description

rcorr: Kendall's tau rank correlation between every pair of a set of variables. Verified against [cor](#)(method = "kendall"): exact match.

Usage

```
tda_rcorr(data, variables = NULL, options = list(), dir = tempfile("tda"), ...)
```

Arguments

<code>data</code>	a data frame or matrix.
<code>variables</code>	optional subset of data's column names to use, instead of all of them.
<code>options</code>	a named list of further TDA options, passed through.
<code>dir</code>	working directory.
<code>...</code>	passed to <code>tda_run</code> .

Value

A matrix, Kendall's tau between each pair of variables, with 1 on the diagonal.

See Also

Other descriptive statistics: `tda_brr()`, `tda_cov()`, `tda_dstat()`, `tda_freq()`, `tda_independence()`, `tda_ineq()`, `tda_loglin()`, `tda_quant()`, `tda_segr()`, `tda_subm()`

Examples

```
set.seed(1)
d <- data.frame(x1 = rnorm(20), x2 = rnorm(20), x3 = rnorm(20))
tda_rcorr(d)
tda_rcorr(d, variables = c("x1", "x2"))
```

tda_rcsv

*Read a CSV file through TDA***Description**

TDA's `rcsv`: parses a csv file and writes a whitespace table. Verified in the suite as a round trip against `read.csv`.

Usage

```
tda_rcsv(file, sep = ";", ...)
```

Arguments

<code>file</code>	csv input. TDA's reader splits on SEMICOLONS (the European dialect); with <code>sep = ","</code> the wrapper converts a comma file on the way in.
<code>sep</code>	the input file's separator.
<code>...</code>	passed to <code>tda_run</code> .

Value

a data frame of TDA’s parse when the export layer is on; otherwise the rendered lines.

With the export layer on (the default), the engine hands back each raw semicolon-bounded line of the converted file and the result is TDA’s parse as a data frame, first line as header. Without it, only TDA’s padded rendering exists; it carries no separator guarantee (adjacent one-character entries can touch), so the rendered lines are returned as they are. The engine accepts a `df=` option for this command but does not write to it.

Examples

```
f <- tempfile(fileext = ".csv")
write.csv(data.frame(a = 1:3, b = c(2.5, 1, 4)), f, row.names = FALSE)
tda_rcsv(f, sep = ",") # a data frame under the export layer
```

tda_rd	<i>Random values, as TDA’s rd()/rdn()</i>
--------	---

Description

The R-side equivalent of what `nvar()`’s `X = rd(0, 3)` or `Y = sin(X) + rd` would generate inside TDA: `tda_rd(n, a,b)` is uniform on `[a, b]` (rd’s two-argument form, `a + random1() * (b - a)`), and `tda_rd(n)` alone – rd with no arguments – is uniform on `[0, 1]`, not the wider, arbitrary range a hand-rolled substitute might reach for. `tda_rdn` is the same idea for `rdn`, a standard normal.

Usage

```
tda_rd(n, a = 0, b = 1)

tda_rdn(n, mean = 0, sd = 1)
```

Arguments

- `n` how many values to draw.
- `a, b` the range, for `tda_rd`; TDA’s default (no arguments) is `[0, 1]`.
- `mean, sd` the normal distribution’s parameters, for `tda_rdn`; TDA’s `rdn` is always standard `(0, 1)`.

Details

These use R’s random number generator, not `tda_runif`’s reproduction of TDA’s exact stream – for building a data frame to pass to TDA (rather than reproducing an existing TDA example’s numbers bit for bit), R’s generator is what `set.seed()` already controls.

Value

A numeric vector of length `n`.

Examples

```
set.seed(1)
x <- tda_rd(200, 0, 3)      # nvar()'s X = rd(0, 3)
y <- sin(x) + tda_rd(200)   # nvar()'s Y = sin(X) + rd
round(c(mean = mean(x), sd = sd(x)), 2)  # near 0 and 3
```

tda_read_dbf	<i>Read a dBase (.dbf) attribute table</i>
--------------	--

Description

tda_read_dbf reads a dBase file through TDA's rdbf and returns it as a data frame. Character fields come back as character, numeric ones as numeric, taken from the field types the file itself declares.

Usage

```
tda_read_dbf(file, options = list(), dir = tempfile("tda"), ...)
```

Arguments

file	path to a .dbf file.
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to tda_run .

Details

This is the attribute half of a shapefile. [tda_read_shapefile](#) reads the geometry; the .dbf beside it holds one row per shape with whatever columns the data has, which for real data is usually where the names live. [tda_read_shapefile](#) attaches it as `attr(x, "attributes")`.

Value

A data frame; `attr(x, "run")` carries the run.

See Also

Other spatial analysis: [plot.tda_spatial\(\)](#), [tda_map\(\)](#), [tda_polygons\(\)](#), [tda_read_shapefile\(\)](#), [tda_read_spatial](#), [tda_sd\(\)](#), [tda_sd_analyses](#), [tda_sd_ps3\(\)](#), [tda_spatial\(\)](#)

Examples

```
# a shapefile's .dbf, which sf ships alongside nc.shp
if (requireNamespace("sf", quietly = TRUE)) {
  f <- sub("[.]shp$", ".dbf", system.file("shape/nc.shp", package = "sf"))
  head(tda_read_dbf(f)[, c("NAME", "AREA")])
}
```

tda_read_ps

*Read TDA's PostScript output***Description**

Parses the PostScript a plot command writes into a table of drawing operations: line segments, polygons, symbols and text, with the graphics state that applied to each.

Usage

```
tda_read_ps(file, which = NULL)
```

Arguments

file	a .ps file written by TDA, or a run whose directory contains one.
which	when file is a run, the name of the PostScript file.

Value

A list with ops, the drawing operations, and bbox, the bounding box TDA declared. Each operation carries the plot command that produced it, taken from the %# annotations TDA writes into the stream.

See Also

Other plotting: [plot.tda_ple\(\)](#), [tda_check_ps\(\)](#), [tda_pl\(\)](#), [tda_pl_arc\(\)](#), [tda_pl_axis\(\)](#), [tda_pl_graph\(\)](#), [tda_pl_hist](#), [tda_pl_panel\(\)](#), [tda_pl_regression\(\)](#), [tda_pl_scatter\(\)](#), [tda_plot_ps\(\)](#), [tda_ps\(\)](#), [tda_ps3\(\)](#)

Examples

```
d <- data.frame(x = 1:5, y = c(2, 4, 5, 8, 9))
p <- tda_ps(d, xlim = c(0, 6), ylim = c(0, 10))
p <- tda_pl_axes(p)
p <- tda_pl_lines(p, "x", "y")
pdf(NULL) # plot() runs the session and draws it; capture the run
p <- plot(p)
dev.off()
ps <- tda_read_ps(p$run, which = p$file)
ps$bbox
ps$ops[[3]]$op
```

tda_read_shapefile	<i>Read a shapefile into a TDA spatial structure</i>
--------------------	--

Description

sdshp reads an ESRI shapefile – the .shp, its .shx index and the .dbf of attributes – and writes TDA's spatial data file, which is what the rest of the spatial commands work on.

Usage

```
tda_read_shapefile(file, dir = tempfile("tda"), ...)
```

Arguments

file	path to the shapefile, with or without the .shp extension. The .shx and .dbf beside it are read too.
dir	working directory for the run.
...	further options for sdshp.

Value

A tda_spatial object, as [tda_spatial](#) returns.

See Also

Other spatial analysis: [plot.tda_spatial\(\)](#), [tda_map\(\)](#), [tda_polygons\(\)](#), [tda_read_dbf\(\)](#), [tda_read_spatial](#), [tda_sd\(\)](#), [tda_sd_analyses](#), [tda_sd_ps3\(\)](#), [tda_spatial\(\)](#)

Examples

```
if (requireNamespace("sf", quietly = TRUE)) {
  nc <- tda_read_shapefile(system.file("shape/nc.shp", package = "sf"))
  print(tda_sd_info(nc))
  plot(nc)
}
```

tda_read_spatial	<i>Read other spatial formats</i>
------------------	-----------------------------------

Description

The same shape as [tda_read_shapefile](#): TDA reads the file and writes its spatial data file, which the rest of the family works on. [tda_read_e00](#) reads an uncompressed ARC/INFO export, [tda_read_gshhs](#) a GSHHS coastline file, and [tda_read_dcw](#) a Digital Chart of the World polygon point file.

Usage

```
tda_read_e00(file, dir = tempfile("tda"), ...)

tda_read_gshhs(
  file,
  level = NULL,
  centre = c("greenwich", "pacific"),
  dir = tempfile("tda"),
  ...
)

tda_read_dcw(file, dir = tempfile("tda"), ...)
```

Arguments

file	path to the input file.
dir	working directory for the run.
...	further options for the command. <code>sde00</code> takes <code>attr</code> for how many attribute variables to keep; <code>sdgshhs</code> takes <code>level</code> to select polygons; <code>sddcwp</code> takes <code>nc</code> to drop centroids.
level	for <code>tda_read_gshhs</code> : which hierarchy levels to keep – 1 land, 2 lake, 3 island in lake, 4 pond in island in lake, and since GSHHG 2.3.0 5 the Antarctic ice front and 6 the Antarctic grounding line. Several may be given, <code>level = c(1, 2)</code> for land and lakes. <code>NULL</code> , the default, keeps all of them; level 1 alone is much the fastest way to draw a coastline. Antarctica appears twice in a 2.3.x file, once as level 5 and once as level 6, and they are about six degrees of latitude apart near the dateline. Keeping all levels draws both, which is why a world map shows two near-parallel southern coastlines; GSHHG intends you to pick one. Neither is an artefact – the ice front runs along roughly 78 degrees south there (the Ross Ice Shelf edge, near-straight) and the grounding line along roughly 84.
centre	for <code>tda_read_gshhs</code> : where the world map is cut. "greenwich" (default) gives the familiar view, longitudes -180..180, cut at the dateline. "pacific" cuts at the Greenwich meridian instead, longitudes 0..360. Polygons spanning whichever seam is in force are split either way.

Details

All three target older formats from GIS's pre-shapefile era, and only shapefile really remains in wide, current use – e00 and DCW were both effectively superseded by it, and DCW's distribution site has been gone for years, which is worth knowing going in:

- `tda_read_e00` works, tested against a real ARC/INFO export (a point coverage).
- `tda_read_dcw` works, tested against a .pnt file rebuilt from a real DCW coverage read via the **sf** package (`st_read()` on the original coverage directory, coordinates regrouped into rings and written out in the plain text form `sddcwp` expects). Coordinates round-trip exactly: the intermediate this reader writes uses `fmt = "24.16"`, not TDA's default, which would cost six digits of each one.

- `tda_read_gshhs` reads both GSHHS header layouts: the pre-2.2 one and GSHHG 2.x, which is what you would download today. The layout is detected from the file itself and the version reported in the run's output.

Straight lines across a world map would come from the dateline; that is handled for you, and there is nothing to configure. GSHHS stores the polygons that span the dateline with longitudes running past 180 (Eurasia reaches 190). Those are cut at the dateline and come back as separate objects, so each piece is drawn where it belongs and nothing is drawn between them; a polygon that does not span it is untouched. The Antarctic coastline, which runs from one edge of the map to the other, is returned as a line rather than a closed area for the same reason – closing it would draw a straight segment back across the world.

This is what the GSHHG maintainers do in their own shapefile distribution, where the dateline-straddling polygons – the Antarctic cap chief among them – are split into east and west components. The native binary files are left unsplit for the reader to handle.

`centre = "pacific"` cuts the map at Greenwich instead, giving longitudes on 0..360; the splitting then happens at that seam rather than at the dateline. Either way no polygon is drawn across the map.

Value

A `tda_spatial` structure.

Further options

Anything else these commands accept can be passed through . . . under TDA's option name. `tda_help("sdshp")`, `tda_help("sde00")`, `tda_help("sddcw")` and `tda_help("sdgshhs")` print the full lists. One worth knowing is `nc`, which tells the DCW reader that the input has no comment lines.

`df=`, `dt da=` and `fmt=` are set by the reader itself: the first two name the intermediate files it writes and reads back, and `fmt` is fixed at "24.16" so that no coordinate is rounded on the way through.

See Also

Other spatial analysis: `plot.tda_spatial()`, `tda_map()`, `tda_polygons()`, `tda_read_dbf()`, `tda_read_shapefile()`, `tda_sd()`, `tda_sd_analyses`, `tda_sd_ps3()`, `tda_spatial()`

Examples

```
# nc.shp ships with sf, so this one runs where sf is installed.
# The .dbf beside the geometry holds the attribute table -- county
# names and areas -- and is attached as $attributes.
if (requireNamespace("sf", quietly = TRUE)) {
  nc <- tda_read_shapefile(system.file("shape/nc.shp", package = "sf"))
  head(nc$attributes[, c("NAME", "AREA")])
}
```

```
## Not run:
# the other readers need files that do not ship with any package
tda_read_e00("world.e00")
tda_read_gshhs("gshhs.f.b", level = 1) # either header layout
tda_read_dcw("polygon.pnt", nc = TRUE)
```

```
## End(Not run)
```

tda_read_spss	<i>Read an SPSS data file</i>
---------------	-------------------------------

Description

tda_read_spss reads an SPSS system file (.sav, TDA's rspss1) or a portable file (.por, TDA's rspss) and returns its data as a data frame. The format is taken from the file extension unless portable says otherwise.

Usage

```
tda_read_spss(
  file,
  portable = grepl("\\.por$", file, ignore.case = TRUE),
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

file	path to a .sav or .por file.
portable	read as a portable (.por) file. Defaults to whether file ends in .por.
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to tda_run .

Details

These two commands were in TDA all along but had no wrapper, and no test: the only case in the suite fed rspss1 a text file and pinned its refusal. Both are now checked against real files written by **readspss**, and both round-trip their values exactly.

String columns *do* come through: rspss/rspss1 create real TDA string variables (type 1) and pdata prints them, so a character column arrives as text rather than a code. This is one of the few ways to get a string variable into TDA at all – [tda_run](#) refuses a character column in a data frame, because TDA's data-frame path has no type for it.

Two consequences worth knowing. TDA renames any variable whose name does not start with a capital (see *Variable names must start with a capital*), so a lowercase SPSS name comes back as VAR1, VAR2, ...; write the file with capitalised names if you need them preserved. And once a string variable exists, TDA's string operators work on it in a later nvar block of the same run – strlen(S), strsp(S) (the alphabetical sort position, matching R's rank), strv(S) and strvp(S,n,m).

For value labels and missing-value codes, use **readspss** directly: TDA keeps the data and the names, so a label or a user-missing code is lost on the way in, and this wrapper does not pretend otherwise.

Value

A data frame, with `attr(x, "run")` carrying the run.

See Also

Other data: [tda_read_xls\(\)](#), [tda_strings\(\)](#)

Examples

```
# readspss can write the fixture, so this runs where it is installed
if (requireNamespace("readspss", quietly = TRUE)) {
  f <- file.path(tempdir(), "example.sav")
  readspss::write.sav(data.frame(ID = 1:3, X = c(1.5, 2.5, 3.5),
                                   S = c("a", "bb", "ccc")), f)

  tda_read_spss(f)
}
```

 tda_read_stata | *Read a Stata file* |

Description

Reads a Stata `.dta` file through TDA's `rstata`, which supports releases 4, 6, 7 and 10 – the same set [tda_write_stata](#) writes, so a file written here reads back here.

Usage

```
tda_read_stata(
  file,
  n_records = NULL,
  missing = NULL,
  upper_names = FALSE,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

<code>file</code>	path to the <code>.dta</code> file.
<code>n_records</code>	number of records to read. <code>NULL</code> , the default, reads all of them.
<code>missing</code>	value to substitute for Stata's system missing. TDA's default is <code>-5</code> .
<code>upper_names</code>	translate variable names to upper case (<code>rstata</code> 's <code>n=2</code>). Default <code>FALSE</code> .
<code>options</code>	a named list of further TDA options, passed through. The reader also accepts, through options, TDA's <code>noc=</code> , <code>msys=</code> , <code>arcd=</code> , <code>dfa=</code> , <code>fn=</code> , <code>p=</code> , <code>pn=</code> , <code>zoo=</code> and <code>vdf=</code> switches.
<code>dir</code>	working directory.
<code>...</code>	passed to tda_run .

Value

A data frame. `attr(x, "run")` carries the run.

See Also

[tda_write_stata](#), [tda_read_spss](#)

Examples

```
d <- data.frame(id = 1:4, y = c(1.5, 2, 2.5, 3))
f <- tempfile(fileext = ".dta")
tda_write_stata(d, f)
tda_read_stata(f)
```

 tda_read_sys | *Read a TDA system file* |**Description**

Reads a system file written by [tda_write_sys](#). A system file carries the data matrix and the variable definitions together, so the frame comes back with its column names without an `nvar` block being repeated.

Usage

```
tda_read_sys(file, options = list(), dir = tempfile("tda"), ...)
```

Arguments

<code>file</code>	path to the system file.
<code>options</code>	a named list of further TDA options, passed through. The underlying pdata print accepts, through options, its selection and format switches: <code>keep=</code> , <code>drop=</code> , <code>sort=</code> , <code>noc=</code> , <code>nn=</code> , <code>nc=</code> , <code>nq=</code> , <code>ap=</code> , <code>l0=</code> , <code>sepc=</code> , <code>sd=</code> and <code>sdfmt=</code> .
<code>dir</code>	working directory.
<code>...</code>	passed to tda_run .

Value

A data frame. `attr(x, "run")` carries the run.

See Also

[tda_write_sys](#)

Examples

```
d <- data.frame(id = 1:4, y = c(1.5, 2, 2.5, 3))
f <- tempfile(fileext = ".sys")
tda_write_sys(d, f)
tda_read_sys(f)
```

tda_read_table	<i>Read TDA's output files</i>
----------------	--------------------------------

Description

tda_read_table reads one table, taking column names from the comment header where they can be matched to the columns. tda_read_blocks splits a file holding several tables, which is what a life table writes when there are groups. tda_file and tda_blocks do the same relative to a run's directory, and report what TDA said if the file is missing.

Usage

```
tda_read_table(file, col_names = NULL)

tda_read_blocks(file, col_names = NULL)

tda_blocks(x, file, ...)

tda_file(x, file, ...)
```

Arguments

file	a file name.
col_names	column names to apply; a list for tda_read_blocks, one entry per block.
x	a run or fitted model.
...	passed on.

Value

A data frame, or a list of them.

See Also

Other TDA infrastructure: [tda_ccnt\(\)](#), [tda_help\(\)](#), [tda_output\(\)](#), [tda_run\(\)](#), [tda_write_data\(\)](#)

Examples

```
set.seed(39)
d <- data.frame(income = round(rlnorm(30, 8, 0.5)))
fit <- tda_ineq(d)
# tda_file() reads a named output file relative to the run's directory
tda_file(fit, "out.txt", col_names = c("index", "cases", "min", "max",
                                       "mean", "sd", "vcoeff", "gini"))

# tda_blocks(): a life table with groups writes several tables into one
# file -- counts, then a survivor/density/rate table, once per group
set.seed(1)
d2 <- data.frame(t = c(4, 3, 1, 5, 8, 2), s = c(1, 1, 0, 1, 1, 1),
                 g = c(1, 1, 1, 2, 2, 2))
lt <- tda_ltb(Surv(t, s) ~ as.factor(g), d2, tp = seq(0, 10, 2))
length(tda_blocks(lt, "out.ltb")) # 4: two groups, two tables each

# tda_read_blocks(): the same thing from a raw file path, when there is
# no fit object to resolve one from
path <- file.path(lt$run$dir, "out.ltb")
length(tda_read_blocks(path))
```

tda_read_ucinet

Read a UCINET file

Description

Reads a UCINET dataset through TDA's `rucinet`. UCINET stores a dataset as a pair of files, a header `##h` and the data `##d`; name the one you want read.

Usage

```
tda_read_ucinet(
  file,
  n_columns = NULL,
  form = c("matrix", "edges"),
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

<code>file</code>	path to the UCINET file, usually the <code>##d</code> .
<code>n_columns</code>	number of columns. <code>NULL</code> , the default, leaves TDA to work it out.
<code>form</code>	"matrix" (default) reads it as a square matrix; "edges" reads it as an edge list, keeping only entries greater than zero.
<code>options</code>	a named list of further TDA options, passed through.
<code>dir</code>	working directory.
<code>...</code>	passed to <code>tda_run</code> .

Details

begins a comment in TDA's command language, so a file called something.##d truncates the command that names it and TDA reports "Command file ends with an incomplete command". The file is therefore copied to a #-free name for the run – which is why this takes a path rather than leaving the caller to discover that.

Value

A data frame of the values TDA read. `attr(x, "run")` carries the run.

Examples

```
# a Ucinet ##d file is packed 4-byte floats, row-major
M <- rbind(c(0, 2, 0), c(2, 0, 1), c(0, 1, 0))
f <- tempfile(fileext = "##d")
con <- file(f, "wb")
writeBin(as.numeric(t(M)), con, size = 4); close(con)
tda_read_ucinet(f, n_columns = 3)
tda_read_ucinet(f, n_columns = 3, form = "edges")
```

tda_read_xls

Read an Excel .xls workbook

Description

`tda_read_xls` reads a BIFF .xls workbook through TDA's `rxls` and returns the numeric table it extracts. Every sheet is stacked into one frame with a leading sheet column.

Usage

```
tda_read_xls(
  file,
  skip_unknown = TRUE,
  numeric_only = FALSE,
  protocol = FALSE,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

<code>file</code>	path to a .xls file.
<code>skip_unknown</code>	skip record types TDA does not know (default TRUE; see above).
<code>numeric_only</code>	keep only columns TDA classed as numerical (TDA's <code>ns=1</code>).
<code>protocol</code>	also return the workbook's string table, as <code>attr(x, "strings")</code> .

options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to tda_run .

Details

Unknown record types are skipped by default (`skip_unknown = TRUE`, TDA's `ni=1`). Excel 2007 and later write records into BIFF8 that TDA's table does not list – XFCRC (0x087c), STYLEEXT (0x0892) and others – so with TDA's default any .xls saved this century stops at “unknown record type” before reaching a single cell. A BIFF record carries its length, so skipping one is safe; naming them individually is not, because the list keeps growing. Pass `skip_unknown = FALSE` for TDA's original behaviour.

Text columns come back as codes. A column of text becomes an integer index into the workbook's string table, not the text. TDA reports which columns those are (the “Column / New / Type” table in the run's output), and `protocol = TRUE` writes the strings themselves. This is TDA's design, not a fault, but it means a column can come back as plausible small integers that are codes: the `cyl` column of **readxl**'s `datasets.xls` is stored as text and arrives as 1, 2, 3 where the values are 4, 6, 8. For anything where that matters, **readxl** reads the workbook directly.

Value

A data frame; `attr(x, "run")` carries the run.

See Also

Other data: [tda_read_spss\(\)](#), [tda_strings\(\)](#)

Examples

```
# readxl ships these workbooks, so this runs where readxl is installed
if (requireNamespace("readxl", quietly = TRUE)) {
  x <- tda_read_xls(readxl::readxl_example("datasets.xls"))
  # three sheets of different widths, stacked with a `sheet` column;
  # each is also available with its own column types
  head(attr(x, "sheets")[[1]])
}
```

 tda_recode | *Recode variables in place* |

Description

TDA's recode: replaces variable values by expressions over the current matrix. Verified in the suite against plain R recoding.

Usage

```
tda_recode(data, ..., options = list())
```

Arguments

data	data frame.
...	named recode expressions in TDA syntax, e.g. <code>x = "x * 2"</code> ; further arguments to <code>tda_run</code> go through options.
options	list passed to <code>tda_run</code> .

Value

the recoded data frame.

Examples

```
d <- data.frame(age = c(17, 25, 40), inc = c(1, 2, 3))
# TDA's expression language: ge() is >=, so age becomes an adult flag
tda_recode(d, age = "ge(age, 18)", inc = "inc * 1000")
```

tda_repsel	<i>Repeat or select block cases</i>
------------	-------------------------------------

Description

TDA's `repsel` with a matrix expression giving, per block, the repetition count.

Usage

```
tda_repsel(data, by, times, commands = character(), ...)
```

Arguments

data	data frame.
by	block-defining column.
times	matrix expression (TDA syntax) of dimension blocks x 1.
commands	further command lines run under the replicated case structure. Column names are TDA-sanitised (upper-cased) first, so refer to them in that form.
...	passed to <code>tda_run</code> .

Value

the printed output, invisibly.

Examples

```
# blocks by G, block 1 twice and block 2 once: cases become
# X = 1,2,1,2,3, and the statistics confirm it (mean 1.8, sum 9)
out <- tda_repsel(data.frame(G = c(1, 1, 2), X = 1:3), by = "G",
                  times = "<2, 1>", commands = "dstat = X;")
grep("^X", out, value = TRUE)
```

tda_rfit

*Fit binary relations to a valued graph***Description**

TDA's `rfit`: finds the relations with the requested properties closest to the graph, via linear programming.

Usage

```
tda_rfit(
  d,
  properties = c("symmetric", "transitive"),
  max_solutions = 10,
  variant = c("standard", "experimental"),
  ...
)
```

Arguments

<code>d</code>	symmetric matrix (proximities).
<code>properties</code>	character subset of "reflexive", "symmetric", "antisymmetric", "transitive", "complete".
<code>max_solutions</code>	cap on the number of equally good relations TDA keeps (<code>max=</code> , 10). When more exist the fit is refused with an error naming this argument; a linear-order request (antisymmetric, transitive, complete) on a symmetric input has 17 optimal relations for 4 nodes and 491 for 6. What comes back is correct but not always complete: against every relation on 4 nodes enumerated in R (test-gap-wrappers.R, and 30 directed cases in the session-52 notes) <code>rfit</code> 's value is always the optimum and every relation it lists is optimal with the properties, but in about one case in five with ties it lists a subset of the tied optima (2 of 4, 4 of 5) – a limit of the CACM 449 enumeration it uses.
<code>variant</code>	"standard" runs <code>rfit</code> ; "experimental" runs the <code>rfit1</code> sibling (same syntax, alternative implementation).
<code>...</code>	passed to <code>tda_run</code> .

Value

relations, one fitted relation matrix per solution, and output.

Examples

```
d <- rbind(c(0, 2, 1), c(2, 0, 2), c(1, 2, 0))
r <- tda_rfit(d)
r$relations[[1]]
```

tda_rmod

*Simple Rasch model***Description**

TDA's rmod on binary item response data.

Usage

```
tda_rmod(x, ...)
```

Arguments

`x` matrix of 0/1 responses, persons in rows.
`...` passed to `tda_run`.

Value

centered item parameters in `$items` (joint ML is identified only up to a constant, and its item estimates carry the known upward bias of about $k/(k-1)$ for k items), the pattern parameters in `$patterns`, and the maximised log likelihood in `$logLik`.

Examples

```
set.seed(1)
ab <- rnorm(80)
x <- sapply(c(-1, 1), function(d) rbinom(80, 1, plogis(ab - d)))
tda_rmod(x)$items    # easier item first, harder second
```

tda_rng

*TDA's random number generator***Description**

`tda_rng` is a stateful generator reproducing TDA's `rd/rdn` bit for bit. `tda_runif/tda_rnorm` are convenience wrappers around a fresh generator, for when only one of `rd/rdn` is needed; `tda_rng` itself is what a `nvar()` block using *both* together needs, since they draw from one continuous, shared stream, not two independent ones.

Usage

```
tda_rng(seed = 13421773)

tda_runif(n, seed = 13421773)

tda_rnorm(n, seed = 13421773)
```

Arguments

seed	the starting value. TDA's default is 13421773.
n	how many numbers to draw, for tda_runif/tda_rnorm.

Details

random1() is the multiplicative generator behind rd(): the seed is multiplied by 3125 modulo 2^{26} , in three steps of 25, 25 and 5. normal() (behind rdn(); rdn1 is a different pseudo-variable, backed by a different function, normal1(), not this one) draws random1() values in a rejection loop (Marsaglia's polar method, J.R. Bell's ACM Algorithm 334) and returns *two* standard normal deviates per accepted draw, the second cached and returned on the next call with no further random1() draws consumed – so the number of underlying uniform draws per rdn() call varies, and cannot be precomputed or batched independently of rd() calls interleaved with it.

This is why nvar()'s evaluation order matters: it evaluates one case fully, one variable at a time in definition order, before moving to the next case – not all of one variable's values before the next. A block defining $X = rd$ then $E = rdn$ draws X 's case 1, then E 's case 1 (however many random1() calls that costs), then X 's case 2, and so on – tda_runif(256) followed separately by tda_rnorm(256) does not reproduce this at all, even with the same seed, since it draws all of X before touching E 's own share of the stream. Reproducing that exact case-by-case interleaving is what tda_rng's stateful \$rd()/rdn() are for: call them in the same order, one case at a time, and the shared state advances exactly as TDA's evaluation loop would.

It matters because several of the shipped examples select a subsample with isel = le(rd(0, 1), p), or generate simulated data with rd/rdn together (doc/npreg1.cf's worked example, say). Reproducing their numbers needs the same stream, and R's generators will not do – this is not a question of setting a seed. tda_runif/tda_rnorm reproduce random1()/normal() bit for bit.

Value

tda_rng returns a list with two functions, \$rd(n = 1, a = 0, b = 1) and \$rdn(n = 1), each drawing n values from the shared stream (a single value by default, for calling one at a time in an interleaved loop) and returning a length-n numeric vector (or a single number, for the default n = 1). \$rd()'s a/b are the range – TDA's rd(a, b); \$rd() alone is [0, 1), matching rd with no arguments. tda_runif/tda_rnorm return a plain numeric vector of length n directly.

Examples

```
tda_runif(5)

# X = rd, E = rdn in the same nvar() block -- interleaved, not two
# separate streams; compare doc/npreg1.cf's worked example.
gen <- tda_rng()
n <- 256
X <- E <- numeric(n)
for (i in seq_len(n)) {
  X[i] <- gen$rd()
  E[i] <- 0.32 * gen$rdn()
}
```

tda_rod	<i>Rank-order data utilities</i>
---------	----------------------------------

Description

TDA's rod: rank orders per record, with options for standardization, distances, medians, and tabulation.

Usage

```
tda_rod(  
  x,  
  statistic = c("standard", "distances", "central", "graph", "graph5", "graph6",  
    "between", "table"),  
  ...  
)
```

Arguments

x	matrix of rank values, one order per row.
statistic	one of "standard", "distances", "central", "graph", "graph5", "graph6", "between", "table".
...	passed to tda_run .

Value

the rank orders as a data frame in \$orders (plus a count column for the table statistic), with the full protocol in \$output.

Examples

```
r <- tda_rod(rbind(c(1, 2, 3), c(2, 3, 1), c(3, 1, 2)))  
r$orders
```

tda_rplz	<i>Read a PLZ-style CSV file</i>
----------	----------------------------------

Description

TDA's rplz: the postcode-data variant of the semicolon CSV reader; like [tda_rcsv](#) it renders to the protocol.

Usage

```
tda_rplz(file, ...)
```

Arguments

file the input file.
... passed to `tda_run`.

Value

a data frame, one row per record: id, type, place count, the three leading fields, and the place list.

Examples

```
# the PLZ export shape: a header line, then three semicolon fields
# of metadata before the place list (commas separate places, colons
# start sub-lists)
f <- tempfile()
writeLines(c("plz;land;kreis;orte",
            "01067;SN;Dresden;Dresden,Altstadt",
            "01069;SN;Dresden;Dresden,Suedvorstadt"), f)
tda_rplz(f) # one row per record, places joined
```

tda_rrdat	<i>The job-episode data from Blossfeld and Rohwer</i>
-----------	---

Description

rrdat.1 is the data file used by nearly every example in `examples/ehhnew`, the command files accompanying Blossfeld and Rohwer, *Techniques of Event History Modeling*. This reads it and names its columns as the examples do.

Usage

```
tda_rrdat(file = NULL, states = c(2, 4), upward = c("gt", "ge"))
```

Arguments

file path to `rrdat.1`. Defaults to the copy shipped with the package.
states 2 for a single destination state, 4 to distinguish upward, lateral and downward job changes.
upward with `states = 4`, how an upward move is coded: "gt" is `ed2.cf's gt(PRESN/PRES - 1, 0.2)`, "ge" is `rt1m.cf's ge(...)`; one episode lies on the boundary, so the two give 84/219 and 85/218 upward/lateral moves. Both use TDA's comparison tolerance.

Value

A data frame of 600 episodes.

Source

The data are 600 job episodes of 201 randomly selected respondents from the German Life History Study (GLHS), collected at the Max-Planck-Institut für Bildungsforschung, Berlin, in 1981-1983 for the birth cohorts 1929-31, 1939-41 and 1949-51 (Mayer and Brückner 1989). TDA's manual (section 3.3.3) thanks Karl Ulrich Mayer and Hans-Peter Blossfeld for providing the data set; it has shipped with TDA since 1994 as `rrdat.1` and is the example throughout the manual and the book.

References

Mayer, K. U., Brückner, E. (1989). *Lebensverläufe und Wohlfahrtsentwicklung. Konzeption, Design und Methodik der Erhebung von Lebensverläufen der Geburtsjahrgänge 1929-1931, 1939-1941, 1949-1951*. Berlin: Max-Planck-Institut für Bildungsforschung.

Blossfeld, H.-P., Rohwer, G. (2002). *Techniques of Event History Modeling. New Approaches to Causal Analysis*, 2nd ed. Mahwah, NJ: Lawrence Erlbaum.

The derived variables are the ones the examples define in their `nvar` blocks:

TFP duration of the episode, $TFin - TStart + 1$.

DES destination state. With `states = 2` it is 1 for a job change and 0 for a censored episode, which is what most of the examples use. With `states = 4` the job changes are split by prestige: 1 upward, 2 lateral, 3 downward.

LFX labour force experience before the episode, $TStart - TE$.

PNOJ number of previous jobs, $NOJ - 1$.

COHO2, COHO3 birth cohort indicators, 1939-41 and 1949-51.

Examples

```
d <- tda_rrdat()
tda_ltb(Surv(TFP, DES) ~ 1, d, tp = seq(0, 500, by = 30))
```

 tda_rsys | *Read a TDA system file* |

Description

TDA's `rsys`: restores a data matrix written by `wsys`. Verified in the suite as a write/read round trip.

Usage

```
tda_rsys(file, ...)
```

Arguments

<code>file</code>	the system file.
<code>...</code>	passed to <code>tda_run</code> .

Value

the restored data as a data frame, like `tda_read_sys`, which this calls.

Examples

```
d0 <- data.frame(id = 1:3, x = c(1.5, 2, 3))
dr <- tempfile("tda"); dir.create(dr)
write.table(d0, file.path(dr, "d.dat"),
            row.names = FALSE, col.names = FALSE)
invisible(tda_run(c("nvar(dfile=d.dat, ID[4.0]=c1, X[8.2]=c2);",
                    "wsys = t.sys;"), dir = dr))
tda_rsys(file.path(dr, "t.sys")) # the values restored
```

tda_run	<i>Run TDA commands</i>
---------	-------------------------

Description

The escape hatch: anything the typed interfaces do not cover can be written as TDA commands and run directly. `tda_run` writes a data frame out and runs a script against it; `tda_run_cf` runs an existing command file.

Usage

```
tda_run(
  commands,
  data = NULL,
  dir = tempfile("tda"),
  data_file = "data.dat",
  ...
)

tda_run_cf(
  file,
  maxnv = 2000L,
  maxmat = 200L,
  args = character(),
  echo = FALSE,
  data = NULL
)
```

Arguments

- `commands` a character vector of TDA commands.
- `data` optional data frame, written to `data_file` in `dir` so that the commands can read it with `dfile=`.
- `dir` working directory; TDA's output files are left there.

data_file	name for the written data file.
...	passed to tda_run_cf.
file	an existing command file.
maxnv, maxmat	limits passed to TDA at start-up.
args	extra command line arguments.
echo	print TDA's output as it is read.

Value

An object of class `tda_result`: `$output` and `$stderr` (the captured text), `$dir`, `$cf` and `$files` (the working directory, the command file, and the files the run produced), `$commands` (the command file's lines, kept on the object), `$diagnostics` (every error, warning or note line TDA printed, see `tda_diagnostics`), `$exports` (TDA's numbers, handed over as matrices in parallel with the printed output), `$errors` (how many errors TDA reported) and `$status` (TDA's exit code, which is 0 for a run that reported errors as well – test `$errors`, not this).

See Also

Other TDA infrastructure: [tda_ccnt\(\)](#), [tda_help\(\)](#), [tda_output\(\)](#), [tda_read_table\(\)](#), [tda_write_data\(\)](#)

Examples

```
tda_run("mem;")

# tda_run_cf: the same thing, but the command file already exists on disk
f <- tempfile()
writeLines("mem;", f)
r <- tda_run_cf(f)
cat(r$output, sep = "\n")
```

tda_scla

Separable clusters of a valued graph

Description

TDA's `scla` on a distance matrix.

Usage

```
tda_scla(d, ...)
```

Arguments

d	symmetric distance matrix.
...	passed to tda_run .

Value

clusters, the rows TDA wrote (one numeric vector per line of the separable-cluster listing), and output.

Examples

```
d <- as.matrix(dist(c(0, 0.1, 5, 5.1)))
r <- tda_scla(d)
r$clusters
```

tda_sd

*Run a spatial command***Description**

tda_sd reaches any of TDA's spatial commands; the named functions are the ones with an obvious meaning. The spatial data is opened with sdnvar first, which is what the commands operate on.

Usage

```
tda_sd(s, cmd, options = list(), rhs = NULL, setup = character(), ...)
```

Arguments

s	a tda_spatial .
cmd	the command name, e.g. "sdfnf", "sdvd".
options	a named list of options for the command.
rhs	the command's right-hand side, when it takes one.
setup	extra commands to run before it, such as a PostScript setup.
...	passed to tda_run .

Value

An object carrying the run and, where the command writes one, a table.

See Also

Other spatial analysis: [plot.tda_spatial\(\)](#), [tda_map\(\)](#), [tda_polygons\(\)](#), [tda_read_dbf\(\)](#), [tda_read_shapefile\(\)](#), [tda_read_spatial](#), [tda_sd_analyses](#), [tda_sd_ps3\(\)](#), [tda_spatial\(\)](#)

Examples

```
d <- data.frame(id = 1:4, x = c(0, 1, 1, 0), y = c(0, 0, 1, 1))
s <- tda_spatial(d)
r <- tda_sd(s, "sdfnf")
cat(r$run$output, sep = "\n")
```

tda_sd_analyses	<i>Spatial analyses</i>
-----------------	-------------------------

Description

One function per spatial command, each a thin wrapper: the command name, the options it takes and the table TDA writes. They all work on a [tda_spatial](#) structure, however it was built – from a data frame with `tda_spatial`, from a shapefile with [tda_read_shapefile](#), or from line segments with [tda_polygons](#).

Usage

```
tda_sd_info(s, ...)

tda_sd_relations(s, ...)

tda_sd_neighbours(s, ...)

tda_sd_enclosing(s, shape = c("rectangle", "hull"), ...)

tda_sd_voronoi(s, ...)

tda_sd_select(s, ...)

tda_sd_intersect(s, ...)

tda_sd_lines(s, ...)

tda_sd_points(s, ...)

tda_sd_data(s, ...)

tda_sd_clip(s, rec, ...)
```

Arguments

<code>s</code>	a tda_spatial .
<code>...</code>	options for the command.
<code>shape</code>	for <code>tda_sd_enclosing</code> , "rectangle" (default) or "hull" (the convex hull); <code>sdenc1</code> 's <code>opt=</code> .
<code>rec</code>	the clipping rectangle, <code>xmin,ymin,xmax,ymax</code> .

Details

`tda_sd_info` reports what a structure holds: how many points, lines and polygons, and the bounding box. `tda_sd_voronoi` builds a Voronoi diagram and the Delaunay triangulation of the points;

tda_sd_enclosing the enclosing rectangle or the convex hull, tda_sd_neighbours nearest-neighbour lists, and tda_sd_relations the relations between objects. tda_sd_select picks out objects in a region – like tda_sd_clip, it needs rec (as an option: tda_sd_select(s, rec = "0,0,1,1")), which TDA will not guess. The polygon operations – tda_sd_intersect, tda_sd_lines, tda_sd_points and tda_sd_clip – each write a record file that comes back as a table.

Value

An object carrying the run and, where there is one, a table.

See Also

Other spatial analysis: [plot.tda_spatial\(\)](#), [tda_map\(\)](#), [tda_polygons\(\)](#), [tda_read_dbf\(\)](#), [tda_read_shapefile\(\)](#), [tda_read_spatial](#), [tda_sd\(\)](#), [tda_sd_ps3\(\)](#), [tda_spatial\(\)](#)

Examples

```
# four points, as a spatial structure
d <- data.frame(id = 1:4, x = c(0, 1, 1, 0), y = c(0, 0, 1, 1))
s <- tda_spatial(d)

# what is in it
tda_sd_info(s)

# the Voronoi diagram of those points: what sdvd writes is itself a
# spatial data file (points for the Voronoi vertices, lines for the
# edges), so plot() reads it back with the input's plot machinery
v <- tda_sd_voronoi(s, opt = 2)
plot(v, xlim = c(-0.5, 1.5), ylim = c(-0.5, 1.5))

# clip to a rectangle; rec is required, TDA will not guess one
tda_sd_clip(s, c(0, 0, 0.5, 0.5))

# a select needs one too, the region to select within
tda_sd_select(s, rec = "0,0,1,1")$table

# relations and nearest neighbours, on the same points
tda_sd_relations(s)
tda_sd_neighbours(s)
tda_sd_enclosing(s, shape = "hull")$table

# polygon data: typ = 3 groups consecutive rows sharing an id into one
# polygon each -- here two squares
poly <- data.frame(id = c(1, 1, 1, 1, 2, 2, 2, 2),
                  x = c(0, 2, 2, 0, 3, 5, 5, 3),
                  y = c(0, 0, 2, 2, 0, 0, 2, 2))
sp <- tda_spatial(poly, type = 3)
tda_sd_data(sp)$table # the polygons' own vertex coordinates back out

# tda_sd_intersect/tda_sd_lines/tda_sd_points compare polygons against a
# *different* object type (points or lines) already in the same spatial
# structure -- with only the two polygons above and nothing else, there
```

```

# is nothing for them to find, which is what they correctly report
tda_sd_lines(sp)$table
tda_sd_intersect(sp)$table
tda_sd_points(sp)$table

# a real shapefile, if one is available: sf's example data, North
# Carolina counties -- everything above works on synthetic points or two
# squares; this is what these look like on real, complicated polygons
if (requireNamespace("sf", quietly = TRUE)) {
  nc <- tda_read_shapefile(system.file("shape/nc.shp", package = "sf"))
  print(tda_sd_info(nc))                # 108 counties, 2421 vertices

  # relations' real answer is printed, not written to a file -- $text
  # holds the boundary comparisons TDA's console output shows
  head(tda_sd_relations(nc)$text, 4)

  # a neighbour network needs line data; a set of polygons alone
  # legitimately has none to report -- 0 nodes, 0 edges, not a bug
  tda_sd_neighbours(nc)$text

  tda_sd_enclosing(nc)$table            # the whole state's bounding box

  # select/clip write one record per vertex, not all the same length --
  # a polygon's header row, then that many coordinate pairs, the
  # same ragged shape as sdvd's edge list above -- a proper rectangular
  # table is what tda_sd_data() gives instead
  tda_sd_select(nc, rec = "-79,35,-77,36")$table # counties in a region
  tda_sd_clip(nc, c(-79, 35, -77, 36))$table     # the same, cut to it
  head(tda_sd_data(nc)$table)                  # every county's vertices,
                                                # as an ordinary data frame
}

```

tda_sd_ps3

Three-dimensional spatial plots

Description

sdplot31, sdplot32 and sdplot33 draw spatial objects in a three-dimensional coordinate system rather than the flat one plot() uses. That system comes from psetup3, so the ranges of all three axes and the direction the plot is viewed from are set up once with tda_sd_ps3(), the 3-D counterpart of [tda_ps](#) – tda_sd_pl3_points(), tda_sd_pl3_lines() and tda_sd_pl3_polygons() then add drawing commands to it the way [tda_pl_lines](#) and its siblings add to a tda_ps session, and plot() draws the result. [tda_pl_text3](#) and [tda_pl_points3](#) (TDA's 3-D-aware commands, not tied to a spatial file) work on a tda_sd_ps3 session too, which is how a title or an extra marker gets added – TDA's ordinary plabel() ([tda_pl_labels](#)) is 2-D only and refuses outright with “current coordinate system is 3-dimensional” if tried here. [tda_pl_lines3](#), by contrast, does **not** work on a tda_sd_ps3 session: it folds its coordinates into a regular nvar variable, which needs an ordinary nvar declaration that a spatial session, declared with sdnvar instead, does not have – “Syntax error or undefined variables”. Use tda_pl_points3 instead, which takes literal coordinates and needs no such declaration.

Usage

```
tda_sd_ps3(
  s,
  xlim = NULL,
  ylim = NULL,
  zlim,
  view = c(30, 30),
  width = 100,
  ...
)

tda_sd_pl3(p, cmd, ..., zvar = NULL)

tda_sd_pl3_points(p, symbol = 1, size = NULL, zvar = NULL, zval = NULL, ...)

tda_sd_pl3_lines(p, symbol = NULL, size = NULL, zvar = NULL, zval = NULL, ...)

tda_sd_pl3_polygons(
  p,
  symbol = NULL,
  size = NULL,
  zvar = NULL,
  zval = NULL,
  ...
)

tda_sd_plot3(
  s,
  z,
  xlim = NULL,
  ylim = NULL,
  view = c(30, 30),
  width = 100,
  newpage = TRUE,
  symbol = 1,
  size = NULL,
  zvar = NULL,
  zval = NULL,
  ...
)

tda_sd_plot3_lines(
  s,
  z,
  xlim = NULL,
  ylim = NULL,
  view = c(30, 30),
  width = 100,
```

```

    newpage = TRUE,
    symbol = NULL,
    size = NULL,
    zvar = NULL,
    zval = NULL,
    ...
)

tda_sd_plot3_polygons(
  s,
  z,
  xlim = NULL,
  ylim = NULL,
  view = c(30, 30),
  width = 100,
  newpage = TRUE,
  symbol = NULL,
  size = NULL,
  zvar = NULL,
  zval = NULL,
  ...
)

```

Arguments

<code>s</code>	a <code>tda_spatial</code> structure.
<code>xlim, ylim</code>	ranges of the x and y axes, taken from the structure when not given.
<code>view</code>	direction of the projection, as longitude and latitude in degrees; TDA's default is 30,30.
<code>width</code>	horizontal size of the plot in mm.
<code>...</code>	further options for the command.
<code>p</code>	a <code>tda_sd_ps3</code> session.
<code>cmd</code>	the TDA command name.
<code>zvar</code>	an attribute carried in from the source file, used as each point's or polygon's height instead of a flat <code>zval</code> – for the polygon commands, each polygon is raised into a prism of that height rather than drawn flat. Name it as it appears in the object record – for a shapefile, a column of the <code>.dbf</code> .
<code>symbol</code>	marker symbol to draw at each point, as TDA's <code>s=</code> . It is named <code>symbol</code> rather than <code>s</code> because <code>s</code> is the spatial object.
<code>size</code>	marker size in mm.
<code>zval</code>	a single height for every point or polygon, when they do not have their own via <code>zvar</code> ; TDA's default is 0.
<code>z, zlim</code>	the range of the z axis, as two numbers. Unlike x and y there is nothing in the spatial file to take it from, so it has no default.
<code>newpage</code>	start a new grid page.

Details

A north-up view of geographic data (longitude/latitude, the way an ordinary map looks) is `view = c(-90, 90)`, not TDA's own default of `c(30, 30)`. This is a property of TDA's azimuth/elevation convention itself, not of any particular dataset, so it is the same for any longitude/latitude data.

Each of the three drawing commands only draws one kind of object, silently drawing nothing at all for any other kind: `tda_sd_pl3_points` draws points, `tda_sd_pl3_lines` draws lines, `tda_sd_pl3_polygons` draws polygons. A shapefile of counties, say, is polygons – points on it draws nothing, not because anything is wrong, but because it is looking for points that are not there. Check [tda_sd_info](#) first if it is not obvious which kind a structure holds.

`tda_sd_plot3`, `tda_sd_plot3_lines` and `tda_sd_plot3_polygons` are one-shot equivalents – set up a session, add one drawing command, plot it – kept for the simple case where nothing else needs adding to the same plot.

Value

`tda_sd_ps3()` and the `tda_sd_pl3_*` functions return the session, with the command added. `plot()` on a session, and the one-shot functions, draw the plot and return the parsed drawing operations invisibly.

See Also

Other spatial analysis: [plot.tda_spatial\(\)](#), [tda_map\(\)](#), [tda_polygons\(\)](#), [tda_read_dbf\(\)](#), [tda_read_shapefile\(\)](#), [tda_read_spatial](#), [tda_sd\(\)](#), [tda_sd_analyses](#), [tda_spatial\(\)](#)

Examples

```
set.seed(1)
d <- data.frame(id = 1:8, x = runif(8), y = runif(8), z = runif(8))
s <- tda_spatial(d, attributes = "z")
pdf(NULL)

# composable form: build a session, add commands, plot it -- a title
# (via TDA's 3-D-aware text command, not the 2-D-only plabel) and an
# extra marker layered onto the same plot as the data
p <- tda_sd_ps3(s, zlim = c(0, 1))
p <- tda_sd_pl3_points(p, symbol = 1, size = 3)
p <- tda_pl_text3(p, "eight points", at = c(0.5, 0.5, 0.9))
plot(p)

# the one-shot form, for when nothing else needs adding
tda_sd_plot3(s, z = c(0, 1), symbol = 1, size = 3)

# plot3_lines wants line-type spatial data, not scattered points --
# a single connected line here, four vertices sharing one id
ln <- data.frame(id = 1, x = c(0, 0.3, 0.6, 1), y = c(0, 0.4, 0.5, 1),
  z = c(0, 0.3, 0.6, 1))
sl <- tda_spatial(ln, type = 2, attributes = "z")
tda_sd_plot3_lines(sl, z = c(0, 1), symbol = 1)

# plot3_polygons draws flat polygons at a single height in the z range,
```

```

# or, given zvar, raises each one into a prism of that height -- h here
poly <- data.frame(id = 1, x = c(0, 2, 2, 0), y = c(0, 0, 2, 2),
                  h = c(2, 2, 2, 2))
sp <- tda_spatial(poly, type = 3, attributes = "h")
tda_sd_plot3_polygons(sp, z = c(0, 3), zvar = "h")

# a real shapefile, if one is available: sf's example data, North
# Carolina counties, in a proper north-up view, with city markers and
# labels layered on via TDA's 3-D-aware point/text commands
if (requireNamespace("sf", quietly = TRUE)) {
  nc <- tda_read_shapefile(system.file("shape/nc.shp", package = "sf"))
  cities <- data.frame(
    name = c("Raleigh", "Charlotte"),
    lon = c(-78.6382, -80.8431), lat = c(35.7796, 35.2271))

  pnc <- tda_sd_ps3(nc, zlim = c(0, 1), view = c(-90, 90))
  pnc <- tda_sd_pl3_polygons(pnc)
  # fs= is the marker/font size in mm, independent for each command --
  # the default marker size (2mm) reads as a large blob at this map's
  # own small default width (100mm), so it is turned down here, and
  # the default text size turned up, rather than using both defaults
  pnc <- tda_pl_points3(pnc, cities$lon, cities$lat, rep(0.5, 2),
                      symbol = 5, size = 0.8, lty = 0)
  for (i in seq_len(nrow(cities)))
    pnc <- tda_pl_text3(pnc, cities$name[i],
                      at = c(cities$lon[i] + 0.35, cities$lat[i], 0.5),
                      fs = 4)

  plot(pnc)
}
dev.off()

```

tda_sddf

*Distribution function of a set-valued discrete variable***Description**

The set-valued counterpart of `tda_iddf`: each case's observation is a *set* of possible categories (not necessarily a single value, and not necessarily an interval), and the distribution function is only identified up to bounds consistent with every case's own set.

Usage

```

tda_sddf(
  sets,
  categories = NULL,
  self_consistent = FALSE,
  control = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)

```

Arguments

<code>sets</code>	a list, one element per case, each the vector of category labels that case's set contains – <code>list(1, 2, 3, c(1, 2), c(2, 3))</code> for the five cases in TDA's manual example ($o_1 = \{1\}$, ..., $o_5 = \{2, 3\}$). Converted internally to the indicator columns <code>sddf</code> itself needs (one 0/1 column per category, 1 if that case's set contains it), the shape TDA's manual example uses. A data frame or matrix of 0/1 indicator columns (one column per category, exactly <code>sddf</code> 's shape – what a plain <code>read.table()</code> import of a file like TDA's <code>id1.dat</code> already is) is accepted directly too, with no conversion needed.
<code>categories</code>	the full list of possible categories; by default, the sorted union of everything appearing in <code>sets</code> .
<code>self_consistent</code>	also compute the self-consistent distribution (<code>opt=2</code>), not just the min/max/mean bounds (<code>opt=1</code> , default) – <code>sddf</code> 's <code>opt=</code> .
<code>control</code>	convergence settings (<code>mxit=/tolf</code> via tda_control), only meaningful with <code>self_consistent = TRUE</code> .
<code>options</code>	a named list of further TDA options, passed through.
<code>dir</code>	working directory.
<code>...</code>	passed to tda_run .

Value

An object carrying a table of categories and the distribution function's bounds and mean at each.

See Also

Other interval-valued data: [tda_bounds\(\)](#), [tda_idf\(\)](#), [tda_ilsreg\(\)](#), [tda_imean\(\)](#), [tda_imreg\(\)](#), [tda_inpreg\(\)](#), [tda_ivar1\(\)](#), [tda_ivreg\(\)](#)

Examples

```
# TDA's manual example: o1={1}, o2={2}, o3={3}, o4={1,2}, o5={2,3}
fit <- tda_sddf(list(1, 2, 3, c(1, 2), c(2, 3)))
fit$table

# the same data, already read in as indicator columns (e.g. from
# id1.dat via read.table()) -- no conversion needed
id1 <- data.frame(X1 = c(1, 0, 0, 1, 0), X2 = c(0, 1, 0, 1, 1),
                  X3 = c(0, 0, 1, 0, 1))
tda_sddf(id1)$table
```

tda_segr	<i>Segregation measures between two groups</i>
----------	--

Description

TDA's `segr`: a dissimilarity D-Index, a variance ratio and a Gini coefficient between two groups, across one or more categorical class variables. The D-Index is $\frac{1}{2} \sum_i |p_{1i} - p_{0i}|$, the share of one group that would have to change class for the two groups' distributions across classes to match exactly.

Usage

```
tda_segr(
  data,
  group,
  variables = NULL,
  options = list(),
  dir = tempfile("tda")
)
```

Arguments

<code>data</code>	a data frame.
<code>group</code>	the grouping variable, as a name or a vector: coded 0 and 1 (or FALSE/TRUE).
<code>variables</code>	one or more categorical class variables from <code>data</code> (by name) to compute the measures for, one row of output each. Defaults to every other column in <code>data</code> when <code>group</code> is itself given as a column name (nothing to default to if <code>group</code> was a raw vector instead, since <code>data</code> then has no column of its own to exclude).
<code>options</code>	a named list of further TDA options, passed through.
<code>dir</code>	working directory.

Value

An object carrying a table: one row per variable in `variables`, with the number of classes, cases in each group, and the three measures.

See Also

Other descriptive statistics: [tda_brr\(\)](#), [tda_cov\(\)](#), [tda_dstat\(\)](#), [tda_freq\(\)](#), [tda_independence\(\)](#), [tda_ineq\(\)](#), [tda_loglin\(\)](#), [tda_quant\(\)](#), [tda_rcorr\(\)](#), [tda_subm\(\)](#)

Examples

```
set.seed(1)
n <- 100
d <- data.frame(group = rbinom(n, 1, 0.5),
                 class1 = sample(1:3, n, replace = TRUE),
```

```
class2 = sample(1:2, n, replace = TRUE))
tda_sogr(d, group = "group", variables = c("class1", "class2"))
```

tda_seq_info

Information about currently defined sequence data structures

Description

Runs TDA's seq; against the sequence structure(s) given – the same table TDA itself prints after defining them, listing each structure's number of state variables, its time axis, and the distinct states it actually contains.

Usage

```
tda_seq_info(
  sequences,
  data = NULL,
  options = list(),
  type = 1,
  dir = tempfile("tda"),
  ...
)
```

Arguments

sequences	a data frame or matrix, one row per case, one column per time point, in order – the same shape tda_seqm takes – or, given data too, a character vector naming those columns within it directly, rather than requiring a separately pre-subsetted data frame: real sequence data typically carries an ID column and covariates alongside the states in the same data frame (TDA's examples/exam/seq.d4 , e.g.), not only the states on their own. For tda_seqev/tda_seqevd , also a <i>list</i> of several such specs, to define several independent sequence data structures at once – TDA's <code>seqdef(sn=1)</code> , <code>seqdef(sn=2)</code> , ... – the pattern TDA's own manual uses to build a second sequence structure from <code>seq.d4</code> 's <code>S0..S5</code> columns, entirely separate from the first (<code>Y0..Y5</code>), not a covariate of it. Use <code>sn=</code> to say which one a given call analyzes.
data	optional data frame sequences (and, for tda_seqmd , <code>covariates</code>) name columns within, instead of sequences/covariates being the actual data already.
options	a named list of further TDA options, passed through.
type	the sequence data type, TDA's <code>seqdef m=</code> 1 (default) one column per time point, 2 columns read as state/time pairs, one pair per spell.
dir	working directory.
...	passed to tda_run .

Value

A data frame, one row per sequence structure: `sn`, `type`, `variables` (number of time points), `tmin/tmax` (the time axis), `nstates`, and `states` (a list column, the actual state values, since their count varies by structure).

See Also

Other sequence analysis: [tda_evalfi\(\)](#), [tda_fml\(\)](#), [tda_frml\(\)](#), [tda_seqgc\(\)](#), [tda_seqm\(\)](#), [tda_seqmd\(\)](#), [tda_seqpe\(\)](#), [tda_seqpm\(\)](#)

Examples

```
s <- data.frame(Y0 = c(1, 1, 2), Y1 = c(1, 2, 1), Y2 = c(2, 1, 1))
tda_seq_info(s)

# several structures at once, the same way tda_seqev's sn= does
d <- data.frame(Y0 = c(1, 1, 2), Y1 = c(1, 2, 1), S0 = c(1, 3, 1),
               S1 = c(3, 1, 3))
tda_seq_info(list(c("Y0", "Y1"), c("S0", "S1")), data = d)
```

tda_seqgc

*Sequence characteristics, lengths, gaps and state distributions***Description**

Descriptive summaries of a sequence data structure: `tda_seqgc` basic characteristics per case (length, number of states visited, number of transitions, time spent in and number of episodes in each state), `tda_seqlg` sequence lengths and gaps, `tda_seqs` the distribution of states over time (how many cases are in each state at each time point), `tda_seqs_i` a state indicator matrix (one row per case-time combination, one column per state, 1 where that case is in that state at that time), and `tda_seqen` entropy of the state distribution at each time point. Column names come from TDA's own description of each output, not a fixed layout guessed in advance – `tda_seqgc`'s columns, for instance, depend on how many distinct states the data actually has.

Usage

```
tda_seqgc(
  sequences,
  select = NULL,
  variables = NULL,
  id = NULL,
  options = list(),
  type = 1,
  dir = tempfile("tda"),
  ...
)
```

```

tda_seqlg(
  sequences,
  select = NULL,
  variables = NULL,
  id = NULL,
  options = list(),
  type = 1,
  dir = tempfile("tda"),
  ...
)

tda_seqsdi(
  sequences,
  select = NULL,
  variables = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)

tda_seqsdi(
  sequences,
  tp,
  select = NULL,
  variables = NULL,
  id = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)

tda_seqen(
  sequences,
  select = NULL,
  variables = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)

```

Arguments

sequences	a data frame or matrix of one column per time point, in order, the same shape tda_seqm takes – optionally with further, non-sequence columns (an id, say) alongside them; use <code>variables/id</code> to say which is which rather than pre-subsetting sequences yourself.
select	optional expression (<code>sel=</code>) restricting which sequences are used – written with R’s comparison operators (<code>Y0 == 1</code> , or <code>Y0 = 1</code>) or TDA’s function form (<code>eq(Y0, 1)</code>)

	interchangeably; &/ combine several conditions the same way in both.
variables	which columns of sequences are the actual time-point columns, in order – TDA's seqdef=Y0,,Y7, which likewise names the sequence out of a data matrix holding other variables too rather than requiring them to be dropped first. A character vector of names in sequences; defaults to every column (so passing an already-subsetted sequences still works exactly as before).
id	further columns of sequences to carry through to the output as-is, one value per case – seqgc/seqlg/ seqpm/seqsi's v=, which writes the case's raw value onto its output row, no more. A character vector of names in sequences. seqsd and seqen have no per-case row to attach a value to (one row per time point, over all cases) and error if given one – TDA accepts v= there without complaint but never actually writes the column.
options	a named list of further TDA options, passed through.
type	the sequence data type, TDA's seqdef m=: 1 (default) one column per time point, 2 columns read as state/time pairs, one pair per spell, with the time axis taken from the times rather than from the column count.
dir	working directory.
...	passed to tda_run .
tp	for tda_seqsi, the time points to build the indicator matrix at – required, as a vector or a single TDA range expression such as "0(1)10".

Value

A data frame, one row per case (tda_seqgc, tda_seqlg) or per time point (tda_seqsd, tda_seqen), or per case-time combination (tda_seqsi).

See Also

Other sequence analysis: [tda_evalfi\(\)](#), [tda_fml\(\)](#), [tda_frml\(\)](#), [tda_seq_info\(\)](#), [tda_seqm\(\)](#), [tda_seqmd\(\)](#), [tda_seqpe\(\)](#), [tda_seqpm\(\)](#)

Examples

```
s <- data.frame(Y0 = c(1, 1, 2), Y1 = c(1, 2, 1), Y2 = c(2, 1, 1),
               Y3 = c(1, 1, 2), Y4 = c(1, 1, 2), Y5 = c(3, 3, 3))
tda_seqgc(s) # length, states visited, transitions, time in each state
tda_seqlg(s) # lengths and gaps
tda_seqsd(s) # how many cases are in each state, at each time point
tda_seqsi(s, tp = "0(1)5") # state indicator matrix
tda_seqen(s) # entropy of the state distribution at each time point

# an id column alongside the sequence, positively selected rather
# than requiring s2[, -1]
s2 <- cbind(id = 101:103, s)
tda_seqgc(s2, variables = names(s), id = "id")
```

tda_seqm

*Sequence analysis by optimal matching***Description**

Computes proximities between sequences and returns a dist, so hclust, cmdscale and the rest work on the result directly.

Usage

```
tda_seqm(
  sequences,
  method = 1,
  indel = NULL,
  subcost = NULL,
  common_length = FALSE,
  max_restrict = NULL,
  compare_with = NULL,
  preprocess = NULL,
  n_random = NULL,
  print = NULL,
  dp_matrix = FALSE,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

sequences	one row per case, one column per time point – a data frame or matrix, or a list of vectors of differing lengths, one per case, padded with NA to a common width internally (TDA's seqdef accepts NA for a case's unused trailing positions). States may be numeric, character or factor; the alphabet is taken over the whole frame.
method	optimal matching method, TDA's m=.
indel	indel cost, TDA's icost=.
subcost	substitution costs, TDA's scost=: either a single number selecting one of TDA's built-in cost schemes, or a full k x k matrix of costs (k = number of distinct states in sequences), which gets written out as an mdef() block and referenced by name (TDA itself only accepts a matrix <i>name</i> for scost=, never a literal list of numbers). Rows and columns follow the states in ascending order, TDA's internal ordering – see states in the return value for exactly what that order is for a given call.
common_length	require all sequences to share one length before aligning them (seqm's rr=1); TDA's manual text for this is brief ("use common sequence length"), so this is documented no further than that.

max_restrict	restrict the alignment search (seqm's max=); TDA's manual text for this is brief ("alignment restriction"), so this is documented no further than that.
compare_with	compare every sequence against specific sequences rather than every pair (seqm's cn=, one or more case numbers); TDA's manual text for this is brief ("compare with specified sequences"), so this is documented no further than that.
preprocess	one or both of "skip_gaps" (ignore internal gaps – missing values – during alignment) and "skip_identical" (collapse runs of identical states before aligning) – seqm's sm=.
n_random	request an approximate random sample of this many sequence pairs rather than comparing every pair – seqm's own r=.
print	add further columns to attr(x, "pairs") beyond the default index/length/distance: "sequential" replaces the single distance column with a running, cumulative-so-far cost at each time point (dist_t1..dist_tn, n = number of time points in sequences), also built into attr(x, "dist_by_time"), an n x n x n_time array – [, , t] is the full pairwise distance matrix using only the alignment cost accumulated up to time point t, the same symmetric, zero-diagonal shape x itself has. distance itself becomes the final time point, dist_tn, kept for compatibility with the default shape. "lcs" adds, alongside the usual single distance, the longest common subsequence's own length (lcs_length) and its state sequence, one column per time point (lcs_t1..lcs_tn, -1-padded past the LCS's length). "sequential" errors if compare_with is also given, its shape there not being one this wrapper parses; "lcs" with compare_with runs but leaves its extra columns unnamed for the same reason (one distance/LCS group per reference sequence, not the single group above) – read attr(x, "pairs") (or table for the compare_with shape) directly for these.
dp_matrix	capture TDA's step-by-step dump of the alignment algorithm's internal dynamic-programming matrix (tst=2,3 + df=), one block per pair actually compared – an illustration of how the distance was arrived at, not part of TDA's regular output at all. When TRUE, dp_matrix in the return value is a list keyed "caseA_caseB", each holding seqA/seqB (the two sequences, states in order) and a D matrix (plus E/F for methods with affine costs) – D[i, j] is the alignment cost using only the first i states of seqA and first j of seqB, so D's bottom-right corner is that pair's final distance. Off by default: turning it on can produce a large file, one block per pair compared.
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to tda_run .

Value

Ordinarily a dist, with the pairwise table kept in its pairs attribute, and, with print = "sequential", the per-time-point breakdown also in dist_by_time (see print). Either shape also carries states, the distinct states sequences actually has, in the ascending order subcost (if given as a matrix) must follow, and, with dp_matrix = TRUE, the per-pair alignment breakdown in dp_matrix (see dp_matrix). Also carries run, the underlying [tda_run](#) result – run\$dir is where the generated TDA commands (\$run\$commands) and, with dp_matrix = TRUE, the raw, unparsed debug file TDA

itself wrote (`dp.tst`) both sit, if what `dp_matrix` parsed out of it needs checking directly. With `compare_with` given, there is no full pairwise matrix to return – `cn=` writes each sequence's distance to the reference set instead – so the return value is instead an object carrying that as table (case, length, distance), and states the same way.

Costs

With the default costs a substitution costs twice an indel, so it never beats a delete plus an insert, and the result is the indel distance $|a| + |b| - 2 * \text{LCS}(a, b)$. Setting `indel = 2` makes substitution the cheaper move and the result becomes twice the Levenshtein distance.

See Also

Other sequence analysis: [tda_evalfi\(\)](#), [tda_fml\(\)](#), [tda_frml\(\)](#), [tda_seq_info\(\)](#), [tda_seqgc\(\)](#), [tda_seqmd\(\)](#), [tda_seqpe\(\)](#), [tda_seqpm\(\)](#)

Examples

```
s <- data.frame(t1 = c(1, 1, 2), t2 = c(1, 2, 2), t3 = c(2, 2, 3))
d <- tda_seqm(s)
hclust(d)
```

 tda_seqmd | *Regression models for events* |

Description

Turns sequence data into the discrete-time expanded form a logistic or probit model needs: one row per individual per time point at risk, a binary indicator of whether the target event (a transition from one state to another) occurred there, dummy variables for each time period (so period effects can be estimated freely, the way `factor(t)` would in a regular model), and any covariates carried along. Fit the result with [tda_qreg](#) (`model = "logit" or "probit"`) or [tda_glm](#) (`family = binomial`) on the event column against the period dummies and covariates – TDA's manual (chapter 6.18) calls this approach out explicitly as the discrete-time counterpart to [tda_rate](#).

Usage

```
tda_seqmd(
  sequences,
  event,
  data = NULL,
  covariates = NULL,
  event_covariates = NULL,
  select = NULL,
  tp = NULL,
  summary = FALSE,
  options = list(),
  dir = tempfile("tda"),
```

```

    ...
)

tda_seqev(
  sequences,
  data = NULL,
  sn = 1L,
  select = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)

tda_seqevd(
  sequences,
  data = NULL,
  sn = 1L,
  select = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)

```

Arguments

sequences	a data frame or matrix, one row per case, one column per time point, in order – the same shape <code>tda_seqm</code> takes – or, given data too, a character vector naming those columns within it directly, rather than requiring a separately pre-subsetted data frame: real sequence data typically carries an ID column and covariates alongside the states in the same data frame (TDA's <code>examples/exam/seq.d4</code> , e.g.), not only the states on their own. For <code>tda_seqev</code> / <code>tda_seqevd</code> , also a <i>list</i> of several such specs, to define several independent sequence data structures at once – TDA's <code>seqdef(sn=1)</code> , <code>seqdef(sn=2)</code> , ... – the pattern TDA's own manual uses to build a second sequence structure from <code>seq.d4</code> 's <code>S0..S5</code> columns, entirely separate from the first (<code>Y0..Y5</code>), not a covariate of it. Use <code>sn=</code> to say which one a given call analyzes.
event	for <code>tda_seqmd</code> , the target transition, as <code>c(from, to)</code> – the two states such that going from the first to the second at time <code>t</code> counts as the event.
data	optional data frame sequences (and, for <code>tda_seqmd</code> , covariates) name columns within, instead of sequences/covariates being the actual data already.
covariates	optional further time-independent covariates, carried into the output unchanged – a data frame or matrix, one row per case in the same order as sequences, or (with data given) a character vector naming columns within it.
event_covariates	optional time-varying covariates – TDA's <code>xe=</code> , different from covariates/ <code>v=</code> : a separate value for each time point, rather than one fixed value per case. A named list, needs data: each element the per-time-point column names for one

	such covariate, in the same order as sequences itself, named by what the result's column should be called – <code>list(price = paste0("Price", 0:5))</code> , say, for a covariate stored one column per time point the same way the states themselves are. Exactly how TDA derives the value it puts in the output is not documented – it is not simply the value at the event's time point; check against your data before relying on a particular reading of this column.
select	optional expression (<code>sel=</code>) restricting which sequences are used – written with R's comparison operators (<code>Y0 == 1</code> , or <code>Y0 = 1</code>) or TDA's function form (<code>eq(Y0, 1)</code>) interchangeably; <code>&/ </code> combine several conditions the same way in both.
tp	optional restriction of the time axis, as a vector of time points or a single TDA range expression such as <code>"0(1)10"</code> .
summary	instead of the full per-case, per-period data, just a time/riskset/events table – TDA's <code>seqmd(ev=...)</code> called with no output file at all, a separate mode in which TDA prints this table to the console instead of writing any records. <code>covariates/event_covariates</code> are not accepted together with this, since TDA's console summary carries no covariate columns to show.
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to <code>tda_run</code> .
sn	for <code>tda_seqev/tda_seqevd</code> , which sequence data structure to use when sequences defines more than one (see above) – TDA's <code>sn=</code> , 1 by default, the same as when only one structure was ever defined.

Details

`tda_seqev` and `tda_seqevd` are the exploratory step before committing to a target event: `tda_seqev` counts how often each state-to-state transition actually occurs in the data, and `tda_seqevd` tabulates event counts over time.

A note on names: the manual documents this command as `seqnd`, but TDA 6.4 itself calls it `seqmd`; `tda_seqmd()` uses the name that works.

This wrapper always defines exactly one sequence data structure per call (a fresh `seqdef`, every time), so TDA's `sn=` – selecting among several already-defined structures – has nothing to select among here: `sn = 1` is always correct, and there is no way to reach any other value through this interface. Not exposed as an argument for that reason, rather than included as a parameter that could never meaningfully differ from its default.

`tda_seqev` ignores TDA's `dt da=` (no description file is written even when asked for one); its output shape (`from/to/count`) needs no further description to read reliably. `tda_seqmd`'s `dt da=` is used internally to read TDA's real column structure back, since `event_covariates`, in particular, can change how many columns come back and where.

Value

`tda_seqmd` returns a data frame: `id`, `time`, `event` (the binary indicator), one period dummy column per distinct time point, and any `covariates` – or, with `summary = TRUE`, `time`, `riskset`, `events` instead. `tda_seqev` returns a data frame, one row per transition that occurs in the data: `from`, `to`, `count`. `tda_seqevd` returns a data frame: `time`, `cases`, one match-count column per event type found, and `total`.

See Also

Other sequence analysis: [tda_evalfi\(\)](#), [tda_fml\(\)](#), [tda_frml\(\)](#), [tda_seq_info\(\)](#), [tda_seqgc\(\)](#), [tda_seqm\(\)](#), [tda_seqpe\(\)](#), [tda_seqpm\(\)](#)

Examples

```
# three individuals, six time points (0-5), states 1-3; the target
# event is a 1 -> 2 transition, with two time-independent covariates
s <- data.frame(Y0 = c(1, 1, 2), Y1 = c(1, 2, 1), Y2 = c(2, 1, 1),
               Y3 = c(1, 1, 2), Y4 = c(1, 1, 2), Y5 = c(3, 3, 3))
cov <- data.frame(v1 = c(1, 2, 2), v2 = c(-10, -15, -20))

tda_seqev(s)    # which transitions actually occur, and how often
tda_seqevd(s)   # event counts over time

d <- tda_seqmd(s, event = c(1, 2), covariates = cov)
d
# fit it: a period dummy for every distinct time point, the way
# factor(time) would in an ordinary logistic regression
pd <- grep("^period", names(d), value = TRUE)
tda_qreg(stats::as.formula(paste("event ~", paste(c(pd, "v1", "v2"),
                                                    collapse = "+"))),
         d, model = "logit", intercept = FALSE)

# the same data, but with an ID column and the states/covariates all
# together in one data frame, the more realistic shape -- sequences=
# and covariates= just name columns in data= directly
full <- cbind(id = 1:3, s, cov)
tda_seqmd(paste0("Y", 0:5), event = c(1, 2), data = full,
          covariates = c("v1", "v2"))
```

tda_seqpd

*Write sequence data back out***Description**

seqpd prints the sequence data in one of four layouts, chosen by layout (TDA's m=):

- 1** every sequence structure side by side, one row per case
- 2** one row per case and structure
- 3** one row per case, structure and position
- 4** one row per spell: its origin and destination state, and the times it spans

Usage

```
tda_seqpd(
  sequences,
  layout = 1,
```

```

    select = NULL,
    variables = NULL,
    id = NULL,
    options = list(),
    type = 1,
    second = NULL,
    second_type = 2,
    dir = tempfile("tda"),
    ...
  )

```

Arguments

sequences	a data frame or matrix of one column per time point, in order, the same shape <code>tda_seqm</code> takes – optionally with further, non-sequence columns (an id, say) alongside them; use <code>variables/id</code> to say which is which rather than pre-subsetting sequences yourself.
layout	1 to 4, the shape of the result, as listed above.
select	optional expression (<code>sel=</code>) restricting which sequences are used – written with R’s comparison operators (<code>Y0 == 1</code> , or <code>Y0 = 1</code>) or TDA’s function form (<code>eq(Y0, 1)</code>) interchangeably; <code>&/ </code> combine several conditions the same way in both.
variables	which columns of sequences are the actual time-point columns, in order – TDA’s <code>seqdef=Y0,,Y7</code> , which likewise names the sequence out of a data matrix holding other variables too rather than requiring them to be dropped first. A character vector of names in sequences; defaults to every column (so passing an already-subsetted sequences still works exactly as before).
id	further columns of sequences to carry through to the output as-is, one value per case – <code>seqgc/seqlg/seqpm/seqsi</code> ’s <code>v=</code> , which writes the case’s raw value onto its output row, no more. A character vector of names in sequences. <code>seqsd</code> and <code>seqen</code> have no per-case row to attach a value to (one row per time point, over all cases) and error if given one – TDA accepts <code>v=</code> there without complaint but never actually writes the column.
options	a named list of further TDA options, passed through.
type	the sequence data type, TDA’s <code>seqdef m=</code> : 1 (default) one column per time point, 2 columns read as state/time pairs, one pair per spell, with the time axis taken from the times rather than from the column count.
second	names of the columns of a second sequence structure, declared beside the first as the manual’s <code>seq5.cf</code> does; the tables then carry one column per structure.
second_type	the sequence data type of the second structure, as <code>type</code> ; default 2, state/time pairs.
dir	working directory.
...	passed to <code>tda_run</code> .

Value

A data frame in the requested layout.

Examples

```
# four cases over three states, one sequence structure
s <- data.frame(ID = 1:4,
               Y1 = c(1, 2, 3, 1), Y2 = c(1, 2, 3, 3),
               Y3 = c(2, 2, 1, 3), Y4 = c(2, 3, 1, 1))
tda_seqpd(s, id = "ID")

# one row per spell instead, with the times each spans
tda_seqpd(s, layout = 4, id = "ID")
```

tda_seqpe	<i>Derive a sequence data structure from episode data</i>
-----------	---

Description

seqpe: builds a sequence – a state at every time point – from episode data (a start time, an end time, and origin/destination states for each spell). The result can be read back with a fresh call to [tda_seqm](#) or any of the other sequence functions once written out and re-read; this function only performs the conversion, since TDA's output is a plain data file rather than something kept in memory across calls the way [tda_seqm](#) manages internally.

Usage

```
tda_seqpe(
  data,
  id,
  origin,
  destination,
  start,
  end,
  tp,
  missing = NULL,
  missing_start = NULL,
  missing_end = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

data	a data frame of episodes, one row per spell.
id	name of the case identifier column in data.
origin, destination	names of the origin- and destination-state columns.
start, end	names of the start- and end-time columns.

tp	definition of the time points the sequence is built over, as a vector or a single TDA range expression such as "0(1)10".
missing	value to use for a gap at the beginning (missing_start), at the end (missing_end), or anywhere else (missing); TDA's default is -1 for all three.
missing_start, missing_end	see missing.
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to tda_run .

Value

A data frame: the case id, then one column per time point in tp, each holding the state the case was in at that time.

See Also

Other sequence analysis: [tda_evalfi\(\)](#), [tda_fml\(\)](#), [tda_frml\(\)](#), [tda_seq_info\(\)](#), [tda_seqgc\(\)](#), [tda_seqm\(\)](#), [tda_seqmd\(\)](#), [tda_seqpm\(\)](#)

Examples

```
# two cases, two spells each
e <- data.frame(id = c(1, 1, 2, 2), org = c(1, 2, 1, 3),
               des = c(2, 3, 3, 1), ts = c(0, 3, 0, 2),
               tf = c(3, 5, 2, 5))
tda_seqpe(e, id = "id", origin = "org", destination = "des",
          start = "ts", end = "tf", tp = "0(1)5")
```

tda_seqpm	<i>Sequence pattern matching</i>
-----------	----------------------------------

Description

Counts, per sequence, how many times each of up to 20 patterns occurs – TDA's seqpm. A pattern is a vector of states and wildcards, one element per position: c(3, 3) looks for state 3 immediately followed by state 3; "?" matches any one state, "*" any run of states (including none), "+" any repeat of the state before it, and "-" any run of *identical* states. Several patterns are given as a list, list(c(3, 3), c(3, "*"), 3)).

Usage

```
tda_seqpm(
  sequences,
  patterns,
  variables = NULL,
  id = NULL,
```

```

options = list(),
dir = tempfile("tda"),
...
)

```

Arguments

sequences	a data frame or matrix of one column per time point, in order – the same shape tda_seqm takes – optionally with further, non-sequence columns (an id, say) alongside them; use variables/id to say which is which.
patterns	a list of patterns, each a vector of states (numbers) and wildcards ("?", "*", "+", "-"), as strings), as described above. At most 20.
variables	which columns of sequences are the actual time-point columns, in order – TDA's seqdef=Y0,,Y7, which likewise names the sequence out of a data matrix holding other variables too. A character vector of names in sequences; defaults to every column (so an already-subsetted sequences still works unchanged).
id	further columns of sequences to carry through to the output as-is, one value per case – seqpm's v=. A character vector of names in sequences.
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to tda_run .

Value

An object carrying a table with one row per case: its sequence id, its length, one match-count column per pattern, and any id columns requested.

See Also

Other sequence analysis: [tda_evalfi\(\)](#), [tda_fml\(\)](#), [tda_frml\(\)](#), [tda_seq_info\(\)](#), [tda_seqgc\(\)](#), [tda_seqm\(\)](#), [tda_seqmd\(\)](#), [tda_seqpe\(\)](#)

Examples

```

# six short sequences over three states; does "3,3" (state 3 twice in a
# row) occur, and does the sequence ever return to 1 after leaving it?
s <- data.frame(Y1 = c(1, 1, 3, 3, 3, 1), Y2 = c(1, 3, 3, 3, 1, 2),
               Y3 = c(1, 3, 3, 1, 2, 2), Y4 = c(2, 3, 1, 2, 2, 2),
               Y5 = c(2, 1, 2, 2, 2, 3))
r <- tda_seqpm(s, list(c(3, 3), c(1, "*", 1)))
r$table

# an id column alongside the sequence, positively selected
s2 <- cbind(person = 101:106, s)
r2 <- tda_seqpm(s2, list(c(3, 3)), variables = names(s), id = "person")
r2$table

```

tda_sga	<i>Scalogram (Guttman) analysis</i>
---------	-------------------------------------

Description

TDA's sga: values > 0 mean success on a task.

Usage

```
tda_sga(x, ...)
```

Arguments

`x` matrix, persons in rows, tasks in columns.
`...` passed to `tda_run`.

Value

the printed output.

Examples

```
r <- tda_sga(rbind(c(1, 1, 0), c(1, 0, 0), c(1, 1, 1)))
cat(head(tda_payload(r), 12), sep = "\n")
```

tda_sma	<i>Moving average</i>
---------	-----------------------

Description

A weighted moving average over a set of cases, TDA's sma.

Usage

```
tda_sma(
  x,
  width = NULL,
  weights = NULL,
  options = list(),
  dir = tempfile("tda")
)
```

Arguments

<code>x</code>	the variable to average.
<code>width</code>	an odd window size for an equal-weight moving average, e.g. 3 for the average of each case with its immediate neighbour on each side. Give this or <code>weights</code> , not both.
<code>weights</code>	TDA's half-kernel directly (see above), for an unequal-weight window – <code>c(0.5, 0.25)</code> for a 3-point window weighted 0.25, 0.5, 0.25.
<code>options</code>	a named list of further TDA options, passed through: <code>opt = 1</code> (default) or 2 for how the ends are handled, <code>r=</code> a repeat factor.
<code>dir</code>	working directory.

Details

`gss=` is TDA's weight specification, and it takes only half a symmetric kernel: the centre weight followed by the weights moving out to *one* side, mirrored automatically to build the full window – `c(1, 1)` becomes the three weights 1, 1, 1; `c(1, 1, 1)` becomes five, 1, 1, 1, 1. There is no `tp=` at all: a plain equal-weight window over consecutive cases is what `sma` does, not a smooth over a separate time axis. Weights that sum to 1 give an ordinary average, matching `zoo::rollmean()` for the equal-weight case aside from TDA's own end-value rule where `rollmean` would give NA.

Value

An object carrying a table: the case index, the original value, and the smoothed value.

See Also

Other smoothing: `tda_integrate()`, `tda_interp()`, `tda_isotonic()`, `tda_mat()`, `tda_smd()`, `tda_spl()`

Examples

```
x <- c(1, 2, 3, 10, 5, 6, 7, 8, 9, 10)
tda_sma(x, width = 3)$table      # equal-weight 3-point average
tda_sma(x, weights = c(0.5, 0.25))$table # weighted 0.25, 0.5, 0.25
```

tda_smd

*Robust smoothing by running medians***Description**

A sequence of running-median smoothing passes, resistant to outliers the way a mean is not.

Usage

```
tda_smd(x, sm = "3R", options = list(), dir = tempfile("tda"))
```

Arguments

x	the variable to smooth.
sm	the smoothing operations as a string of characters: "3" is a running median of three, "3R" repeats it to convergence, "2" and "4" are medians of two and four, and they are concatenated to apply in sequence, so "33" is two passes of a 3-median. Not a vector of numbers – a comma is a syntax error.
options	a named list of further TDA options, passed through.
dir	working directory.

Value

An object carrying a table: one row per case, with the raw value beside its smoothed one.

See Also

Other smoothing: [tda_integrate\(\)](#), [tda_interp\(\)](#), [tda_isotonic\(\)](#), [tda_mat\(\)](#), [tda_sma\(\)](#), [tda_spl\(\)](#)

Examples

```
# a running median is robust to the single outlier at position 9
x <- c(5, 2, 8, 3, 9, 1, 7, 4, 100, 6)
tda_smd(x, sm = "3R")$table
```

tda_spatial

Spatial data for TDA's spatial commands

Description

Builds TDA's spatial data structure from a data frame of coordinates. Each object – a point, a line or a polygon – is a group of rows sharing an id.

Usage

```
tda_spatial(
  data,
  id = "id",
  x = "x",
  y = "y",
  attributes = NULL,
  type = 1,
  options = list(),
  dir = tempfile("tda")
)
```

Arguments

<code>data</code>	a data frame with an id column and two coordinate columns.
<code>id, x, y</code>	the column names.
<code>attributes</code>	further columns to carry along, TDA's <code>av=</code> .
<code>type</code>	type of spatial object, TDA's <code>typ=</code> : 1 for points.
<code>options</code>	a named list of further <code>sdgen</code> options.
<code>dir</code>	working directory; the spatial file is written there and stays for the commands that read it.

Value

A `tda_spatial`.

See Also

Other spatial analysis: [plot.tda_spatial\(\)](#), [tda_map\(\)](#), [tda_polygons\(\)](#), [tda_read_dbf\(\)](#), [tda_read_shapefile\(\)](#), [tda_read_spatial](#), [tda_sd\(\)](#), [tda_sd_analyses](#), [tda_sd_ps3\(\)](#)

Examples

```
d <- data.frame(id = 1:4, x = c(0, 1, 1, 0), y = c(0, 0, 1, 1))
s <- tda_spatial(d)
tda_sd_info(s)
```

 `tda_spl` | *Smoothing spline* |**Description**

A smoothing spline through a pair of variables, optionally weighted. `smooth.spline` is the R analogue.

Usage

```
tda_spl(
  x,
  y,
  weights = NULL,
  sig = 1,
  deg = NULL,
  max = 200,
  rx = NULL,
  options = list(),
  dir = tempfile("tda")
)
```

Arguments

<code>x, y</code>	the pair of variables.
<code>weights</code>	optional per-point weights, the same idea as <code>weights</code> on the fitting functions.
<code>sig</code>	smoothing factor; 0 interpolates.
<code>deg</code>	degree of the spline.
<code>max</code>	maximum number of knots.
<code>rx</code>	range over which to evaluate, e.g. <code>seq(0, 10, 1)</code> .
<code>options</code>	a named list of further TDA options, passed through.
<code>dir</code>	working directory.

Value

An object carrying a table of fitted values.

See Also

Other smoothing: [tda_integrate\(\)](#), [tda_interp\(\)](#), [tda_isotonic\(\)](#), [tda_mat\(\)](#), [tda_sma\(\)](#), [tda_smd\(\)](#)

Examples

```
set.seed(33)
x <- sort(runif(30, 0, 10))
y <- sin(x) + rnorm(30, sd = 0.15)
sp <- tda_spl(x, y, sig = 1)
head(sp$table)
```

 tda_split |

 Split episodes at a time-varying covariate |

Description

A covariate that changes during an episode is handled by cutting the episode in two at the moment it changes, so that each piece has one value. TDA does this with the `split` option of `edef`, and [tda_episodes](#) and [tda_rate](#) take a `split` argument that reaches it.

Usage

```
tda_split(
  formula,
  data,
  at = NULL,
  grid = NULL,
  vars = NULL,
  dir = tempfile("tda"),
  ...
)
```

Arguments

<code>formula</code>	the usual <code>Surv()</code> formula.
<code>data</code>	a data frame.
<code>at</code>	one or more variables holding a split time per episode, as names or as a vector, matrix or data frame – one column per cut.
<code>grid</code>	fixed times at which every episode is cut, as a numeric vector. Combines with <code>at</code> .
<code>vars</code>	further variables to carry into the split data.
<code>dir</code>	working directory.
<code>...</code>	passed to <code>tda_run</code> .

Details

`at` names a variable holding, for each episode, the time at which the change happens – measured on the same axis as the episode's start and end. An episode whose covariate never changes is given a value beyond its end, conventionally a large number, and is not cut.

`edef` cuts once per split variable, so several are needed to cut an episode more than once. `grid` is the common case said directly: a set of fixed times every episode is cut at, one split variable per time. `grid = seq(60, 420, by = 60)` is how TDA's `rrdat.d60` was built, 600 job episodes becoming 1021 pieces of at most five years.

Value

A data frame of split episodes, one row per piece.

See Also

Other rate models: `tda_constrain()`, `tda_control()`, `tda_dple()`, `tda_rates()`, `tda_survivor()`, `tda_transitions()`, `vcov.tda_fit()`

Examples

```
d <- tda_rrdat()
# marriage during the job episode, or beyond its end if never
d$MarrDate <- ifelse(d$TMAR <= 0, 10000, d$TMAR - d$TStart)
head(tda_split(Surv(TFP, DES) ~ EDU, d, at = "MarrDate"))
# every five years, however long the episode
nrow(tda_split(Surv(TFP, DES) ~ EDU, d, grid = seq(60, 420, by = 60)))
```

tda_spmod

*Stationary solution of a Leslie matrix***Description**

TDA's spmod: iterates a population vector under the Leslie matrix built from age-specific fertility (first row) and survivor rates (subdiagonal) until the age distribution is stationary. Verified in the suite against R's eigen: the growth factor equals the dominant eigenvalue and the stationary vector the normalized dominant eigenvector.

Usage

```
tda_spmod(
  fertility,
  survival,
  start = rep(1, length(fertility)),
  tolf = 1e-08,
  mxit = 500,
  ...
)
```

Arguments

fertility, survival, start	numeric vectors of equal length: age-specific fertility, survivor rates (last entry unused), and the starting population.
tolf	convergence tolerance.
mxit	iteration cap.
...	passed to tda_run .

Value

list with growth (the dominant eigenvalue) and stationary (the stable age distribution, summing to 1).

Examples

```
r <- tda_spmod(c(0, .4, .3, .1), c(.9, .8, .7, 0), rep(1, 4))
r
L <- rbind(c(0, .4, .3, .1), cbind(diag(c(.9, .8, .7)), 0))
stopifnot(abs(r$growth - Re(eigen(L)$values[1])) < 1e-5)
```

tda_strings	<i>String variables from a run</i>
-------------	------------------------------------

Description

tda_strings returns the string variables a run printed with pdata, as a named list of character vectors.

Usage

```
tda_strings(res, trim = TRUE)
```

Arguments

res	a run, as returned by tda_run .
trim	trim the fixed-width padding (default TRUE).

Details

TDA stores strings fixed-width, so every value is padded to the variable's declared length; the padding is trimmed here, since it is storage rather than data. The values come through the export channel rather than being re-read from the printed file: a string cannot travel on the numeric channel, and parsing it back out of fixed-width text is exactly what the exports exist to avoid.

A character or factor column in a data frame becomes a TDA string variable (see [tda_write_data](#)), and rpsps and rstata create them from the file's string columns. Those are the only ways TDA makes one.

Value

A named list of character vectors, one per string variable, or an empty list if the run printed none.

See Also

Other data: [tda_read_spss\(\)](#), [tda_read_xls\(\)](#)

Examples

```
d <- data.frame(ID = 1:3, S = c("pear", "fig", "date"),
               X = c(1.5, 2.5, 3.5))
r <- tda_run(c(tda_nvar(d), "pdata() = out.txt;"), data = d)
tda_strings(r)
```

tda_subm

*Substitution-metric distance between distributions***Description**

tda_subm measures the substitution-metric distance between a reference distribution and one or more others, TDA's subm.

Usage

```
tda_subm(..., options = list(), dir = tempfile("tda"))
```

Arguments

... the distribution variables: the reference first, then one or more to compare against it.

options a named list of further TDA options, passed through.

dir working directory.

Details

Its columns are not raw data: every value must lie in $[0, 1]$, one row per support point, each column a probability distribution over the same support – TDA rejects anything else with “no distribution in variable”. The first column is the reference; the rest are compared against it, giving one distance each.

TDA prints the distances rather than writing them to the file df= names (that file holds the substitution detail, not the summary), so tda_subm parses them from the run's output into distances, a named numeric vector – table stays NULL unless options = list(df = ...) is given, in which case it holds that detail instead.

Value

An object carrying distances, a named numeric vector.

See Also

Other descriptive statistics: [tda_brr\(\)](#), [tda_cov\(\)](#), [tda_dstat\(\)](#), [tda_freq\(\)](#), [tda_independence\(\)](#), [tda_ineq\(\)](#), [tda_loglin\(\)](#), [tda_quant\(\)](#), [tda_rcorr\(\)](#), [tda_segr\(\)](#)

Examples

```
# six points of support (say, a 6-point agreement scale): Y1 is close
# to the reference X, Y2 is a very different shape
d <- data.frame(X = c(0.05, 0.15, 0.30, 0.30, 0.15, 0.05),
               Y1 = c(0.03, 0.12, 0.32, 0.33, 0.15, 0.05),
               Y2 = c(0.30, 0.30, 0.20, 0.10, 0.07, 0.03))
tda_subm(d)$distances
```

tda_survivor

*The survivor function as a single table***Description**

TDA writes one block per group and its examples select a row range out of the flat file with `tset1 = Case[16, , 30]`. This returns the whole thing as one long data frame instead, with a group column, so it can be plotted, subset or joined without knowing how many blocks there were.

Usage

```
tda_survivor(x, conf.int = 0.95)
```

Arguments

<code>x</code>	a fit from <code>tda_ltb</code> , <code>tda_ple</code> or <code>tda_km</code> .
<code>conf.int</code>	width of the confidence band to add as lower and upper, or NULL for none. TDA has no band of its own – it reports the standard error and leaves the band to the caller – so these are the usual normal limits, clamped to [0, 1].

Details

The same call works for `tda_ltb` and `tda_ple`: both name their time column `time` here, whatever the underlying command called it, and both carry `survivor` and `std.err`. Where the estimator provides them, `density` and `rate` come too.

Value

A data frame with `group`, `time`, `survivor`, `std.err`, and where available `density`, `rate`, `lower` and `upper`.

See Also

Other rate models: `tda_constrain()`, `tda_control()`, `tda_dple()`, `tda_rates()`, `tda_split()`, `tda_transitions()`, `vcov.tda_fit()`

Examples

```
d <- data.frame(t = c(4, 3, 1, 5, 8, 2), s = c(1, 1, 0, 1, 1, 1),
               g = c(1, 1, 1, 2, 2, 2))
f <- tda_ltb(Surv(t, s) ~ as.factor(g), d, tp = seq(0, 10, 2))
tda_survivor(f)
```

tda_tnet	<i>Temporal-network summaries</i>
----------	-----------------------------------

Description

TDA's tnet on a directed edge list.

Usage

```
tda_tnet(  
  edges,  
  report = c("basic", "subgraphs", "indegrees", "outdegrees", "layers"),  
  ...  
)
```

Arguments

edges	data frame: from, to, then at least three value columns in the order VALIDITY (≥ 0 keeps the edge), then the edge's valid-from and valid-to dates with from $\leq$ to – established from the source's check; the command requires a multigraph with three or more subgraphs.
report	"basic", "subgraphs", "indegrees", "outdegrees", or "layers".
...	passed to tda_run .

Value

the printed output.

Examples

```
r <- tda_tnet(data.frame(i = c(1, 2, 3), j = c(2, 3, 4),  
                        valid = c(1, 1, 1),  
                        from = c(10, 12, 14), to = c(20, 22, 24)))  
cat(head(tda_payload(r), 12), sep = "\n")
```

tda_transitions	<i>Transitions covered by a fit</i>
-----------------	-------------------------------------

Description

The origin and destination state pairs a fitted model covers. A multi-state fit returns more than one row; a single-transition fit returns one.

Usage

```
tda_transitions(x)
```

Arguments

`x` a fitted model.

Value

A data frame with `Org` and `Des`, or `NULL`.

See Also

Other rate models: [tda_constrain\(\)](#), [tda_control\(\)](#), [tda_dple\(\)](#), [tda_rates\(\)](#), [tda_split\(\)](#), [tda_survivor\(\)](#), [vcov.tda_fit\(\)](#)

Examples

```
set.seed(9)
mn <- 60
m1 <- data.frame(ts = 0, tf = round(rexp(mn, 0.1), 1) + 0.5, org = 0,
                 des = sample(c(0, 1, 2), mn, TRUE, c(.25, .45, .30)),
                 x = round(rnorm(mn), 2))
mi <- which(m1$des == 1)
m2 <- data.frame(ts = m1$tf[mi],
                 tf = m1$tf[mi] + round(rexp(length(mi), 0.08), 1) + 0.5,
                 org = 1, des = sample(c(1, 2), length(mi), TRUE, c(.4, .6)),
                 x = m1$x[mi])
md <- rbind(m1, m2)
# three transitions: 0->1, 0->2 (censored at 2), 1->2
mf <- tda_rate(Surv(ts, tf, org, des) ~ x, md, model = "exponential")
tda_transitions(mf)
```

tda_triang

*Triangulation of points***Description**

TDA's triang over an x,y point set.

Usage

```
tda_triang(x, y, ...)
```

Arguments

`x, y` coordinate vectors.
`...` passed to [tda_run](#).

Value

the output file's lines (the triangle list).

Examples

```
# a Delaunay triangulation of five points: the edges of the triangles
tda_triangu(x = c(0, 2, 1, 3, 1), y = c(0, 0, 1, 1, 2))
```

tda_ucl

Split an ordering into contiguous clusters

Description

tda_ucl splits an ordering of a graph's nodes into clusters contiguous groups so that the largest cluster diameter is as small as possible, by the dynamic program of Alpert and Kahng (1997). The diameter of a cluster is the largest edge value between any two of its nodes.

Usage

```
tda_ucl(
  g,
  clusters = 2,
  min_size = NULL,
  max_size = NULL,
  order = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

g	a tda_graph , undirected and valued.
clusters	number of clusters, at least 2 (default 2).
min_size, max_size	smallest and largest number of nodes a cluster may have. Defaults: 1, and the number of nodes.
order	the ordering to split, as node indices 1 . . n in the sequence they should be considered. NULL, the default, uses the graph's node order.
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to tda_run .

Details

The clusters are runs of the *ordering*, not arbitrary subsets: this answers "where do I cut this sequence" rather than "which things group together". Give the ordering with *order*; the default is the nodes in their internal order, which is rarely what you want unless the graph was built that way. A seriation or a first principal coordinate is the usual source.

The graph must be undirected and valued, and no edge value may be negative or missing.

Value

A data frame with one row per node: position in the ordering, node (internal index), cluster, and label (the node number the graph was built with). `attr(x, "diameter")` is the largest cluster diameter, the quantity minimised, and `attr(x, "run")` carries the run.

A correctness note

TDA 6.4's ucl did not work. It optimised over the identity ordering with its first two nodes swapped, counted a state with no legal split as costing nothing, searched only the first few split positions, and read past the end of its own cost matrix. On the example shipped with TDA it returned a partition of diameter 7 where one of 4 exists. The number of clusters and the size bounds were fixed at 3, 1 and 3 whatever you asked for.

All of that is repaired here, and the result is checked in this package's tests against a brute force over every contiguous partition. The fixes are ours, not Rohwer's, and are listed in the package's README.

References

Alpert, C. J. and Kahng, A. B. (1997). Splitting an ordering into a partition to minimize diameter. *Journal of Classification* **14**, 51–74.

See Also

Other graph analysis: [plot.tda_graph\(\)](#), [tda_dmet\(\)](#), [tda_g\(\)](#), [tda_g_analyses](#), [tda_graph\(\)](#), [tda_locate_line\(\)](#), [tda_ptree\(\)](#)

Examples

```
# six points on a line, distance growing with the gap between them
d <- as.dist(outer(1:6, 1:6, function(i, j) 3 * abs(i - j) + 1))
g <- tda_graph(d)
p <- tda_ucl(g, clusters = 3)
p
attr(p, "diameter") # the largest cluster diameter, minimised
```

tda_unf

Unfolding of preference data

Description

TDA's unf: each row ranks the alternatives (larger is preferred).

Usage

```
tda_unf(x, ...)
```

Arguments

`x` matrix of preference values.
`...` passed to `tda_run`.

Value

the printed output.

Examples

```
r <- tda_unf(rbind(c(3, 2, 1), c(1, 3, 2), c(2, 3, 1)))
cat(tail(tda_payload(r), 4), sep = "\n") # the best permutation
```

<code>tda_write_data</code>	<i>Build TDA's input</i>
-----------------------------	--------------------------

Description

`tda_write_data` writes a data frame as the whitespace-delimited file TDA reads; `tda_nvar` generates the variable block that describes it; `tda_block` builds any other command. The typed interfaces use these, and they are exported so a hand-written script can too.

Usage

```
tda_write_data(data, file, na = ".")

tda_nvar(
  data,
  file = NA_character_,
  vars = colnames(data),
  mpnt = -5,
  fmt = NULL,
  extra = NULL,
  ...
)

tda_block(name, ..., rhs = NULL)
```

Arguments

`data` a data frame.
`file` output file name.
`na` the marker written for a missing value. It has to be a marker rather than a number: a number is data to TDA, counted and used by the fits. "." is the point marker `nvar` understands.
`vars` variable names, defaulting to the column names.

mpnt	the value the point marker stands for. TDA substitutes it and reports how many there were.
fmt	print formats, one per variable.
extra	further variable definitions, as text.
...	further options, as name = value.
name	a command name.
rhs	the command's right-hand side.

Value

A character vector of commands, or the file name, invisibly.

Numeric precision, storage, and the exports

Every wrapper hands its data over through `rdataframe`, which stores R double columns as TDA `<8>` double arrays and integer columns as `<5>` int arrays – nothing is squeezed through a narrower type on the way in. All of TDA's estimation arithmetic is double, and the direct-export channel (`$run$exports`) copies doubles into R doubles, so an exported value is bit-for-bit the number TDA computed; only the *printed* output rounds, at `tfmt/mfmt`. Two deliberate exceptions to know about: TDA's default storage for variables declared in a hand-written command file (`tda_run(cf=)`, or `tda_nvar` with `file=` and `fmt=`) is a 4-byte float, faithful to the original program – declare `<8>` in the definition if you need double storage there; and a few of TDA's internal summaries (the episode table's weighted counts, for one) pass through float by TDA's design, which the package reproduces rather than corrects.

The `tdaR.use_exports` option

Every reader takes its numbers from the export channel and falls back to parsing TDA's printed output or its written files only if the export is absent or does not fit. Setting `options(tdaR.use_exports = FALSE)` forces that fallback everywhere, which is useful for two things: comparing the two paths when a value looks wrong, and reproducing what an older version of the package returned.

Expect the parser path to give you *less precision*, not different answers – it reads values TDA has already rounded for printing, typically to four or five decimals. A few readers also differ in shape, because a text table cannot always express what the export can: `tda_atab` keeps the open first and last classes (the cases outside your breaks) on the export path and drops them on the parser path, since TDA prints those rows with a blank bound that the fixed-width reader cannot parse.

The package's test suite runs under both settings, so the fallback stays working rather than bit-rotting.

Variable names must start with a capital

TDA's parser accepts a variable name only if it starts with an uppercase letter, `_`, `@` or `$`; a lowercase start is read as some other token. This is not an arbitrary restriction that could be patched out: the first character's case is how TDA's expression lexer tells variables apart from everything spelled in lowercase – functions (`log`, `exp`, `if`), comparison operators (`eq`, `le`, `gt`), the episode accessors (`ts`, `tf`, `org`, `des`, `time`), random draws (`rd`, `rdn`), and the estimated parameters (`b1`, `b2`, ...). Allowing a

lowercase variable would make a column called `time` or `b1` ambiguous against those, with no rule to break the tie, so the case requirement stays and the R side translates instead.

The typed wrappers do that translation invisibly: a lowercase name is sent to TDA as `V<name>` (`.tda_names()`, internal) and the caller's spelling is put back on everything returned – coefficient tables, `vcov()` dimnames, table columns, labels. The `V` spelling appears only in raw TDA console output (`$run$output`) and in command files, where it is TDA's name for the column. Only when writing `tda_nvar/tda_block` commands by hand does the requirement reach you: use names that start with a capital there, or pass them through the same convention.

The symptom varies by exactly where the bad name ends up, and none of the variants mention case, which makes this easy to lose an hour to (it has happened more than once writing this package): a lowercase name used to *reference* an existing variable fails with “Syntax error or undefined variables”, pointing at the command, not the name; a lowercase name used to *declare* a new one – `x = c4` rather than `X = c4` in a hand-built `sdnvar` or `nvar` block, say – fails differently and more confusingly, with “Syntax error: `x=c4,`”, which looks like a problem with the column reference, the trailing comma, or a missing type/format specifier, and is none of those.

String variables and the string operators

A character or factor column becomes a TDA *string variable*: `tda_nvar` declares it with `= str(n,m)` (TDA's only way of making one) and `tda_write_data` pads every field to a fixed width so those character positions are known. Strings are written last in the file whatever order your columns are in, because TDA cannot reach a `a = cK` field lying after a string one; the declaration keeps your order, so nothing changes on the R side. [tda_strings](#) returns them as character.

Four operators work on a string variable, and `extra=` accepts a `{ }` block of plain R, so they read as R rather than TDA syntax:

```
tda_nvar(d, extra = {
  Len = strlen(S)      # storage width, not visible characters
  Rank = strsp(S)      # alphabetical position, as R's rank()
  Num = strv(Code)     # digits to a number, -1 if not all digits
  Sub = strvp(Code, 2, 3) # characters 2..3 as a number
})
```

`strlen` returns the declared storage size, so a padded “fig” in a width-4 variable is 4, not 3. `strv` and `strvp` return -1 when the text is not all digits, rather than failing.

See Also

Other TDA infrastructure: [tda_ccnt\(\)](#), [tda_help\(\)](#), [tda_output\(\)](#), [tda_read_table\(\)](#), [tda_run\(\)](#)

Examples

```
d <- data.frame(Vx = c(1, 2, 3, 4), Vy = c(2, 4, 5, 9))

# the plain data file tda_run() writes and TDA reads -- fixed-width
# columns, no header, exactly what dfile= in an nvar block points at
f <- tempfile()
tda_write_data(d, f)
cat(readLines(f), sep = "\n")
```

```
cmds <- c(tda_nvar(d), tda_block("dstat", rhs = "Vx,Vy"))
cat(cmds, sep = "\n")
res <- tda_run(cmds, data = d)
cat(res$output, sep = "\n")
```

tda_write_foreign	<i>Write a data frame to SPSS, Stata or a TDA system file</i>
-------------------	---

Description

These build TDA's internal data matrix from data and then hand it to TDA's writers. `tda_write_spss` produces a portable file (.por) or, with `format = "sav"`, an SPSS system file. `tda_write_stata` produces a .dta for a chosen Stata release. `tda_write_sys` produces a TDA system file, which [tda_read_sys](#) reads back.

Usage

```
tda_write_spss(
  data,
  file,
  keep = NULL,
  drop = NULL,
  sort = NULL,
  format = c("portable", "sav"),
  options = list(),
  dir = tempfile("tda"),
  ...
)

tda_write_stata(
  data,
  file,
  keep = NULL,
  drop = NULL,
  sort = NULL,
  release = 10,
  options = list(),
  dir = tempfile("tda"),
  ...
)

tda_write_sys(data, file, options = list(), dir = tempfile("tda"), ...)
```

Arguments

`data` a data frame.

file	path to write to.
keep, drop	variables to keep or drop, as a character vector. Give at most one of them.
sort	variables to sort the cases by, as a character vector.
format	for tda_write_spss: "portable" (the default, TDA's wspss) or "sav" (wspss1).
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to tda_run .
release	for tda_write_stata: the Stata release to write for – 4, 6, 7 or 10 (default).

Details

A TDA system file is TDA's format: it stores the data matrix together with the variable definitions, so reading one back restores the session's variables without repeating the nvar block.

Value

file, invisibly. `attr(x, "run")` carries the run.

Variable names

TDA will not read a variable name that does not begin with a capital – `nvar(id = c1)` is a syntax error to it, and the same parser handles keep, drop and sort. Columns are therefore capitalised on the way in, so `id` is written as `Id`. Give keep, drop and sort the names as they are in data; the capitalisation is applied for you.

See Also

[tda_read_spss](#), [tda_read_sys](#)

Examples

```
d <- data.frame(id = 1:5, y = c(2.5, 3, 1.5, 4, 2))
f <- tempfile(fileext = ".por")
tda_write_spss(d, f)
tda_read_spss(f, portable = TRUE)
```

 tda_zreg | *Regression with a censored response* |

Description

`tda_zreg/tda_zreg1` are TDA's `zreg/zreg1`: least squares with a censoring indicator, a Buckley-James estimator where right-censored cases are iteratively reweighted rather than dropped or treated as exact.

Usage

```
tda_zreg(formula, data, censor, options = list(), dir = tempfile("tda"), ...)

tda_zreg1(
  formula,
  data,
  censor,
  dates = NULL,
  options = list(),
  dir = tempfile("tda"),
  ...
)
```

Arguments

formula	a two-sided formula.
data	a data frame.
censor	the censoring indicator, as a name or a vector: TRUE or 1 marks a case whose response is censored, as in <code>survival::Surv</code> . See Details for how this maps to TDA's <code>yw=</code> .
options	a named list of further TDA options, passed through.
dir	working directory.
...	passed to <code>tda_run</code> .
dates	for <code>tda_zreg1</code> , the names of the event-date columns in <code>data</code> – <code>zreg1</code> 's <code>xv=</code> ; each becomes the time-varying indicator <code>date <= t</code> at every step.

Details

`censor` follows R's convention (`survival::Surv()`'s): TRUE or 1 marks a case whose response is censored. TDA's own `yw=` runs the other way – a case counts as *exact* when its indicator is zero – so this is translated for you, the same correction [tda_lsreg1](#) documents for its `cen=`. Sent through unflipped, `zreg` recovers essentially nothing of a known relationship (a near-zero slope); flipped, it recovers it closely, checked against the uncensored least-squares fit.

The iteration often hits `mxit` (default 20) without TDA's tolerance test passing, even once the estimates have stopped moving – `tda_run` then warns “TDA did not converge”. Raising `options = list(mxit = ...)` rarely changes the estimate by more than its last few iterations already did; check by comparing coefficients at two values of `mxit` rather than assuming the warning means the fit is unusable.

`tda_zreg1` is TDA's `zreg1`, the residual-life variant: instead of one fit, it refits the Buckley-James regression of the *remaining* lifetime ($Y - t$) at every time step $t = 0, 1, 2, \dots$ while more than 10 cases remain at risk, and the result is the whole trajectory of coefficients over time. Time-varying covariates enter as *event dates* (`dates`, TDA's `xv=`): at each step, a date column becomes the indicator “has this event happened by t ” (`date <= t`), so its coefficient traces how experiencing the event shifts expected remaining lifetime. It takes the same plain one-row-per-case frame as `tda_zreg`, plus the date columns – the dichotomisation over time *is* the time variation.

Value

An object of class `tda_fit`.

See Also

Other regression: [TDA_FAMILIES](#), [TDA_QRMODELS](#), [tda_freg\(\)](#), [tda_gdf\(\)](#), [tda_llreg\(\)](#), [tda_lsreg\(\)](#), [tda_mlrc_design\(\)](#), [tda_mreg\(\)](#), [tda_nlreg\(\)](#), [tda_npreg\(\)](#)

Examples

```
set.seed(1)
d <- data.frame(x = round(rnorm(50), 2))
d$y <- round(2 - 0.8 * d$x + rnorm(50, sd = 0.4), 3)
d$cen <- as.integer(d$y > 2.2)
d$yc <- pmin(d$y, 2.2)
# zreg iterates to reweight the censored cases; a "did not converge"
# warning here is typical of the method, not a sign the fit is wrong.
# True coefficients are 2 and -0.8, so this should land close to them.
zr <- tda_zreg(yc ~ x, d, censor = "cen")
coef(zr)
head(fitted(zr))
predict(zr, newdata = data.frame(x = c(0, 1)))
```

unzoo

Read and write Zoo archives

Description

unzoo extracts a .zoo archive, and zoo creates one – [unzip](#) and [zip](#) for Zoo rather than Zip. This is tdaR's code, not part of TDA (TDA can only load an archive's contents into its data matrix, via a command that needs a companion .zad description file, and has no general extract or create capability at all).

Usage

```
unzoo(zoofile, files = NULL, list = FALSE, exdir = ".", overwrite = TRUE)
```

```
zoo(zoofile, files, method = 1L, names = basename(files))
```

Arguments

zoofile	path to a .zoo file, or an <code>http(s)://</code> URL, which is downloaded to a temporary file first.
files	paths of the files to add.
list	if TRUE, return a data frame describing the archive's contents instead of extracting anything.
exdir	directory to extract into, created if it does not exist.

overwrite	whether to replace files already present in exdir.
method	packing method for the members: 1 (LZD, the default) or 0 (stored, uncompressed). TDA's arcd reads all of 0, 1 and 2, but other readers of TDA-written archives may not, and LZD is what zoo itself has always produced.
names	the names the members get in the archive, one per file: by default the files' base names. A name may carry a directory, "sub/x.dat", with / as the separator; it must be relative and may not contain ...

Details

unzoo decodes both of Zoo's packing methods, LZD and LZH, ported from TDA's reference implementation – the same code path TDA itself has used since the 1990s to read .zoo archives, so archives TDA itself could read, this reads too. Both methods are tested against a historical TDA archive (deha1.zoo from the TDA teaching pages, kept outside the package as an external test fixture): every decoded byte matches the CRC-16 the original archive itself stored, and separately, every file zoo writes here was extracted correctly, with a passing CRC-16, by the independent, unrelated zoo/unzoo reference tools (<https://github.com/troglobit/zoo>) – not just read back correctly by this same code, which would only prove self-consistency. zoo writes LZD-compressed entries by default, or stored (uncompressed) ones with method = 0, each with its CRC-16 checksum, readable by any Zoo implementation including TDA's.

Member names are Zoo 2.1's: a long file name and a directory name are stored in the directory entry beside the DOS-style 8.3 short name, and both unzoo and zoo use them, so sub/long_name.dat round-trips as itself. unzoo creates the directories it needs under exdir. An archive zoo writes is, field for field, what the reference zoo tool writes for the same members (its header, every directory entry's names and version bytes, the file leaders, the terminal record; only the members' dates differ, which zoo leaves unset), and TDA's own arcd finds a member by its full stored name. TDA's archive description file separates its fields by blanks, so a member TDA is to read may not have a blank in its name.

Zoo can, in principle, keep more than one stored version of a file under the same name (its version-history feature) – unzoo does not collapse or deduplicate by name, so if an archive has several entries called the same thing, all of them come back as separate rows/files rather than only the latest.

Value

For list = TRUE, a data frame with one row per member: name (with its directory, if the entry stores one), method (0 stored, 1 LZD, 2 LZH), size (original, uncompressed), and ok (whether this reads correctly – always TRUE for methods 0/1/2, since those are the only methods Zoo defines). Otherwise, the paths of the files written, invisibly the same as [unzip](#).

Examples

```
archive <- system.file("extdata", "tda.zoo", package = "tdaR")
unzoo(archive, list = TRUE)

ex <- tempfile()
paths <- unzoo(archive, files = "avar.dat", exdir = ex)
readLines(paths)
```

```
# round-trip: write what was just extracted back out, and read it again
out <- tempfile(fileext = ".zoo")
zoo(out, paths)
unzoo(out, list = TRUE)

# names longer than 8.3 and a directory are kept as given
zoo(out, paths, names = "results/avar_first_run.dat")
unzoo(out, list = TRUE)
ex2 <- tempfile()
unzoo(out, exdir = ex2)
list.files(ex2, recursive = TRUE)
## Not run:
unzoo("https://example.com/archive.zoo", exdir = tempfile())

## End(Not run)
```

Index

* TDA infrastructure

- tda_ccnt, [18](#)
- tda_help, [76](#)
- tda_output, [157](#)
- tda_read_table, [209](#)
- tda_run, [220](#)
- tda_write_data, [260](#)

* clustering

- TDA_CLUSTER, [20](#)
- tda_conjoint, [24](#)
- tda_cutree, [29](#)
- tda_dma, [33](#)
- TDA_MDS, [116](#)
- tda_pdatd, [160](#)

* data

- tda_read_spss, [206](#)
- tda_read_xls, [211](#)
- tda_strings, [253](#)

* descriptive statistics

- tda_brr, [17](#)
- tda_cov, [28](#)
- tda_dstat, [39](#)
- tda_freq, [56](#)
- tda_independence, [83](#)
- tda_ineq, [84](#)
- tda_loglin, [98](#)
- tda_quant, [195](#)
- tda_rcorr, [198](#)
- tda_segr, [231](#)
- tda_subm, [254](#)

* episodes

- tda_episodes, [41](#)
- tda_esort, [43](#)

* graph analysis

- plot.tda_graph, [7](#)
- tda_dmet, [35](#)
- tda_g, [60](#)
- tda_g_analyses, [61](#)
- tda_graph, [74](#)

- tda_locate_line, [97](#)

- tda_ptree, [191](#)

- tda_ucl, [258](#)

* interval-valued data

- tda_bounds, [16](#)
- tda_idf, [78](#)
- tda_ilsreg, [80](#)
- tda_imean, [81](#)
- tda_imreg, [82](#)
- tda_inpreg, [85](#)
- tda_ivar1, [92](#)
- tda_ivreg, [93](#)
- tda_sddf, [229](#)

* matrix algebra

- tda_mcel, [110](#)
- tda_mcent, [111](#)
- tda_mch, [112](#)
- tda_mcross, [113](#)
- tda_mdiag, [115](#)
- tda_mev, [119](#)
- tda_mevs, [120](#)
- tda_midf, [121](#)
- tda_midf1, [122](#)
- tda_midf2, [123](#)
- tda_midf3, [124](#)
- tda_minvs, [125](#)
- tda_mkmet, [126](#)
- tda_mkp, [127](#)
- tda_mmul, [133](#)
- tda_mnc, [134](#)
- tda_mnrow, [135](#)
- tda_mpb1, [136](#)
- tda_mpf1, [137](#)
- tda_mpinv, [138](#)
- tda_mpit, [139](#)
- tda_mple, [140](#)
- tda_mpz, [142](#)
- tda_mscal1, [147](#)
- tda_msqrd, [149](#)

- tda_msvd, 151
- tda_mwvec, 153
- * **matrix reshaping**
 - tda_mag, 104
 - tda_mcat, 109
 - tda_mcvec, 114
 - tda_mrsum, 146
 - tda_msort, 148
 - tda_msrow, 150
 - tda_mtrim, 152
- * **optimization**
 - tda_boolean_min, 14
 - tda_mlp, 127
 - tda_mlpi, 129
 - tda_mls, 131
 - tda_mqap, 143
 - tda_mqp, 144
- * **plotting**
 - plot.tda_ple, 8
 - tda_check_ps, 19
 - tda_pl, 162
 - tda_pl_arc, 166
 - tda_pl_axis, 167
 - tda_pl_graph, 169
 - tda_pl_hist, 170
 - tda_pl_panel, 173
 - tda_pl_regression, 175
 - tda_pl_scatter, 177
 - tda_plot_ps, 180
 - tda_ps, 185
 - tda_ps3, 187
 - tda_read_ps, 202
- * **rate models**
 - tda_constrain, 25
 - tda_control, 26
 - tda_dple, 36
 - tda_rates, 198
 - tda_split, 250
 - tda_survivor, 255
 - tda_transitions, 256
- * **regression**
 - TDA_FAMILIES, 47
 - tda_freg, 54
 - tda_gdf, 69
 - tda_llreg, 95
 - tda_lsreg, 100
 - tda_mlrc_design, 130
 - tda_mreg, 145
 - tda_npreg, 155
 - TDA_QRMODELS, 192
 - tda_zreg, 264
- * **sequence analysis**
 - tda_evalfi, 45
 - tda_fm1, 51
 - tda_frml, 57
 - tda_seq_info, 232
 - tda_seqgc, 233
 - tda_seqm, 236
 - tda_seqmd, 238
 - tda_seqpe, 243
 - tda_seqpm, 244
- * **smoothing**
 - tda_integrate, 86
 - tda_interp, 88
 - tda_isotonic, 90
 - tda_mat, 108
 - tda_sma, 246
 - tda_smd, 247
 - tda_spl, 249
- * **spatial analysis**
 - plot.tda_spatial, 10
 - tda_map, 106
 - tda_polygons, 184
 - tda_read_dbf, 201
 - tda_read_shapefile, 203
 - tda_read_spatial, 203
 - tda_sd, 222
 - tda_sd_analyses, 223
 - tda_sd_ps3, 225
 - tda_spatial, 248
- * **utilities**
 - tda_options, 156
- * **zoo archives**
 - unzoo, 266
- cmdscales, 117
- coef.tda_loglin(tda_loglin), 98
- cor, 198
- dist, 160, 161
- eigen, 40
- isoMDS, 117
- lvl, 6
- manual_map(tda_manual_map), 105

- `model.matrix`, 7
- `plot.tda_graph`, 7, 74, 191
- `plot.tda_graph()`, 35, 60, 65, 74, 97, 191, 259
- `plot.tda_ltb`(`plot.tda_ple`), 8
- `plot.tda_ple`, 8
- `plot.tda_ple()`, 19, 165, 167–169, 172, 174, 176, 178, 180, 186, 189, 202
- `plot.tda_rate`(`plot.tda_ple`), 8
- `plot.tda_spatial`, 10
- `plot.tda_spatial()`, 107, 184, 201, 203, 205, 222, 224, 228, 249
- `prcomp`, 33
- `print.tda_loglin`(`tda_loglin`), 98
- `residuals`, 102
- `residuals.tda_loglin`(`tda_loglin`), 98
- `tda_arcd`, 11, 12, 13
- `tda_arcv`, 12
- `tda_arcvc`, 13
- `tda_atab`(`tda_quant`), 195
- `tda_bfc`, 14
- `tda_block`(`tda_write_data`), 260
- `tda_blocks`(`tda_read_table`), 209
- `tda_boolean_min`, 14
- `tda_boolean_min()`, 129, 130, 133, 143, 145
- `tda_bounds`, 16, 80, 92, 93, 95
- `tda_bounds()`, 79, 80, 82, 83, 85, 92, 95, 230
- `tda_brr`, 17, 105, 106
- `tda_brr()`, 28, 40, 57, 84, 85, 99, 196, 199, 231, 254
- `tda_ccnt`, 18
- `tda_ccnt()`, 76, 158, 209, 221, 262
- `tda_check_ps`, 19
- `tda_check_ps()`, 10, 165, 167–169, 172, 174, 176, 178, 180, 186, 189, 202
- `tda_clpyr`, 20
- `TDA_CLUSTER`, 20, 24, 29, 34, 117, 161
- `tda_cluster`, 160
- `tda_cluster`(`TDA_CLUSTER`), 20
- `tda_com`, 22
- `tda_combine_ps`, 22, 186
- `tda_conj`, 23
- `tda_conjoint`, 24
- `tda_conjoint()`, 21, 29, 34, 117, 161
- `tda_constrain`, 25
- `tda_constrain()`, 27, 38, 198, 251, 255, 257
- `tda_control`, 26, 49, 51, 54, 58, 63, 70, 79, 87, 94, 99, 102, 145, 193, 230
- `tda_control()`, 25, 38, 198, 251, 255, 257
- `tda_converged`(`tda_control`), 26
- `tda_corr`, 28
- `tda_corr`(`tda_dstat`), 39
- `tda_cov`, 28
- `tda_cov()`, 17, 40, 57, 84, 85, 99, 196, 199, 231, 254
- `tda_coxph`, 198
- `tda_cutree`, 29
- `tda_cutree()`, 21, 24, 34, 117, 161
- `tda_cwt_norm`, 30
- `tda_dblock`, 31, 106
- `tda_derive`, 31
- `tda_diagnostics`, 32
- `tda_diple`(`tda_dple`), 36
- `tda_dltb`(`tda_dple`), 36
- `tda_dma`, 33, 161
- `tda_dma()`, 21, 24, 29, 117, 161
- `tda_dmet`, 35
- `tda_dmet()`, 8, 60, 65, 74, 97, 191, 259
- `tda_dmet1`, 36
- `tda_dple`, 36
- `tda_dple()`, 25, 27, 198, 251, 255, 257
- `tda_dsplite`(`tda_ccnt`), 18
- `tda_dstat`, 39
- `tda_dstat()`, 17, 28, 57, 84, 85, 99, 196, 199, 231, 254
- `tda_dump`(`tda_ccnt`), 18
- `tda_eigen`, 40
- `tda_ejoin`(`tda_esort`), 43
- `tda_emerge`(`tda_esort`), 43
- `tda_episodes`, 41, 250
- `tda_episodes()`, 44
- `tda_eselect`(`tda_esort`), 43
- `tda_eskip`(`tda_esort`), 43
- `tda_esort`, 43, 106
- `tda_esort()`, 42
- `tda_estimates`(`tda_output`), 157
- `tda_evalf`, 105
- `tda_evalf`(`tda_integrate`), 86
- `tda_evalfi`, 45
- `tda_evalfi()`, 53, 59, 233, 235, 238, 241, 244, 245
- `tda_expand`(`tda_expressions`), 46
- `tda_expressions`, 46
- `TDA_FAMILIES`, 47, 55, 70, 96, 103, 131, 146,

- [156, 194, 266](#)
- `tda_file` (`tda_read_table`), [209](#)
- `tda_fm1`, [49, 51, 54, 55, 57–59](#)
- `tda_fm1()`, [46, 59, 233, 235, 238, 241, 244, 245](#)
- `tda_freq`, [54](#)
- `tda_freq()`, [50, 70, 96, 103, 131, 146, 156, 194, 266](#)
- `tda_freq`, [56](#)
- `tda_freq()`, [17, 28, 40, 84, 85, 99, 196, 199, 231, 254](#)
- `tda_freq1` (`tda_freq`), [56](#)
- `tda_freq2`, [195](#)
- `tda_freq2` (`tda_freq`), [56](#)
- `tda_fm1`, [57](#)
- `tda_fm1()`, [46, 53, 233, 235, 238, 241, 244, 245](#)
- `tda_g`, [60, 61, 97, 115](#)
- `tda_g()`, [8, 35, 65, 74, 97, 191, 259](#)
- `tda_g_aggregate` (`tda_g_analyses`), [61](#)
- `tda_g_analyses`, [8, 35, 60, 61, 74, 97, 191, 259](#)
- `tda_g_backward` (`tda_g_analyses`), [61](#)
- `tda_g_centred` (`tda_g_analyses`), [61](#)
- `tda_g_cliques` (`tda_g_analyses`), [61](#)
- `tda_g_compact` (`tda_g_analyses`), [61](#)
- `tda_g_components`, [74](#)
- `tda_g_components` (`tda_g_analyses`), [61](#)
- `tda_g_cutpoints` (`tda_g_analyses`), [61](#)
- `tda_g_cycles` (`tda_g_analyses`), [61](#)
- `tda_g_dblocks` (`tda_g_analyses`), [61](#)
- `tda_g_dcycles` (`tda_g_analyses`), [61](#)
- `tda_g_degrees` (`tda_g_analyses`), [61](#)
- `tda_g_dot` (`tda_g_analyses`), [61](#)
- `tda_g_edges` (`tda_g_analyses`), [61](#)
- `tda_g_eigen` (`tda_g_analyses`), [61](#)
- `tda_g_flow` (`tda_g_analyses`), [61](#)
- `tda_g_flowcontrol` (`tda_g_analyses`), [61](#)
- `tda_g_gcliques` (`tda_g_analyses`), [61](#)
- `tda_g_independent` (`tda_g_analyses`), [61](#)
- `tda_g_links` (`tda_g_analyses`), [61](#)
- `tda_g_mst` (`tda_g_analyses`), [61](#)
- `tda_g_neighbourhoods` (`tda_g_analyses`), [61](#)
- `tda_g_nodes` (`tda_g_analyses`), [61](#)
- `tda_g_ownership` (`tda_g_analyses`), [61](#)
- `tda_g_paths` (`tda_g_analyses`), [61](#)
- `tda_g_random` (`tda_g_analyses`), [61](#)
- `tda_g_reachable` (`tda_g_analyses`), [61](#)
- `tda_g_shortest` (`tda_g_analyses`), [61](#)
- `tda_g_spanning` (`tda_g_analyses`), [61](#)
- `tda_g_spantrees` (`tda_g_analyses`), [61](#)
- `tda_g_symmetric` (`tda_g_analyses`), [61](#)
- `tda_g_toposort` (`tda_g_analyses`), [61](#)
- `tda_g_transitive` (`tda_g_analyses`), [61](#)
- `tda_gap`, [67](#)
- `tda_gap_permute`, [67](#)
- `tda_gde`, [68](#)
- `tda_gdf`, [69](#)
- `tda_gdf()`, [50, 55, 96, 103, 131, 146, 156, 194, 266](#)
- `tda_gdu`, [70](#)
- `tda_gfcf`, [71](#)
- `tda_ghd`, [72](#)
- `tda_glm`, [30, 238](#)
- `tda_glm` (`TDA_FAMILIES`), [47](#)
- `tda_gmin` (`tda_integrate`), [86](#)
- `tda_gpro`, [72](#)
- `tda_gqap`, [67, 73](#)
- `tda_graph`, [8, 35, 60, 63, 74, 191, 258](#)
- `tda_graph()`, [8, 35, 60, 65, 97, 191, 259](#)
- `tda_gtopo`, [75](#)
- `tda_help`, [76, 156](#)
- `tda_help()`, [18, 158, 209, 221, 262](#)
- `tda_icorr`, [89](#)
- `tda_icorr` (`tda_icov`), [77](#)
- `tda_icov`, [77, 89](#)
- `tda_iddf`, [229](#)
- `tda_iddf` (`tda_idf`), [78](#)
- `tda_idf`, [78](#)
- `tda_idf()`, [16, 80, 82, 83, 85, 92, 95, 230](#)
- `tda_igini` (`tda_imean`), [81](#)
- `tda_ilsreg`, [80, 89](#)
- `tda_ilsreg()`, [16, 79, 82, 83, 85, 92, 95, 230](#)
- `tda_imean`, [81, 89](#)
- `tda_imean()`, [16, 79, 80, 83, 85, 92, 95, 230](#)
- `tda_imreg`, [82](#)
- `tda_imreg()`, [16, 79, 80, 82, 85, 92, 95, 230](#)
- `tda_independence`, [83](#)
- `tda_independence()`, [17, 28, 40, 57, 85, 99, 196, 199, 231, 254](#)
- `tda_ineq`, [84](#)
- `tda_ineq()`, [17, 28, 40, 57, 84, 99, 196, 199, 231, 254](#)
- `tda_inpreg`, [85](#)
- `tda_inpreg()`, [16, 79, 80, 82, 83, 92, 95, 230](#)

- tda_integrate, 86
- tda_integrate(), 88, 91, 108, 247, 248, 250
- tda_interp, 88
- tda_interp(), 87, 91, 108, 247, 248, 250
- tda_interval_wages, 89
- tda_isd, 89
- tda_isotonic, 90
- tda_isotonic(), 87, 88, 108, 247, 248, 250
- tda_ivar, 78, 91, 92
- tda_ivar1, 91, 92
- tda_ivar1(), 16, 79, 80, 82, 83, 85, 95, 230
- tda_ivariance, 77, 89–92
- tda_ivariance(tda_imean), 81
- tda_ivls(tda_ivreg), 93
- tda_ivreg, 16, 80, 89, 93
- tda_ivreg(), 16, 79, 80, 82, 83, 85, 92, 230
- tda_ivreg1(tda_ivreg), 93
- tda_ivreg2(tda_ivreg), 93
- tda_km, 9, 255
- tda_km(tda_dple), 36
- tda_l1reg, 95, 175
- tda_l1reg(), 50, 55, 70, 103, 131, 146, 156, 194, 266
- tda_lcnt(tda_ccnt), 18
- tda_locate_line, 97
- tda_locate_line(), 8, 35, 60, 65, 74, 191, 259
- tda_loglin, 98, 195
- tda_loglin(), 17, 28, 40, 57, 84, 85, 196, 199, 231, 254
- tda_lsreg, 49, 52, 100, 193
- tda_lsreg(), 50, 55, 70, 96, 131, 146, 156, 194, 266
- tda_lsreg1, 69, 265
- tda_lsreg1(tda_lsreg), 100
- tda_ltb, 30, 255
- tda_ltb(tda_dple), 36
- TDA_LTY(tda_pl), 162
- tda_mag, 104
- tda_mag(), 109, 114, 147, 149, 150, 152
- tda_manual_map, 105
- tda_map, 106
- tda_map(), 11, 184, 201, 203, 205, 222, 224, 228, 249
- tda_mat, 108
- tda_mat(), 87, 88, 91, 247, 248, 250
- tda_mcatv, 109
- tda_mcatv(tda_mcatv), 109
- tda_mcel, 110
- tda_mcel(), 111–113, 116, 120–127, 134–139, 141, 142, 148, 149, 151, 153
- tda_mcent, 111
- tda_mcent(), 110, 112, 113, 116, 120–127, 134–139, 141, 142, 148, 149, 151, 153
- tda_mch, 112
- tda_mch(), 110, 111, 113, 116, 120–127, 134–139, 141, 142, 148, 149, 151, 153
- tda_mchol(tda_mchol), 133
- tda_mcross, 113
- tda_mcross(), 110–112, 116, 120–127, 134–139, 141, 142, 148, 149, 151, 153
- tda_mcsun(tda_mrsum), 146
- tda_mcvect, 114
- tda_mcvect(), 105, 109, 147, 149, 150, 152
- tda_mdcent(tda_mcent), 111
- tda_mdcol(tda_mrsum), 146
- tda_mdefg, 106, 115
- tda_mdiag, 115
- tda_mdiag(), 110–113, 120–127, 134–139, 141, 142, 148, 149, 151, 153
- tda_mdiagd(tda_mdiag), 115
- tda_mdrow(tda_mrsum), 146
- TDA_MDS, 21, 24, 29, 34, 116, 161
- tda_mds, 118, 160
- tda_mds(TDA_MDS), 116
- tda_mdsn, 118
- tda_mdsr, 119
- tda_memory(tda_help), 76
- tda_mev, 119
- tda_mev(), 110–113, 116, 121–127, 134–139, 141, 142, 148, 149, 151, 153
- tda_mevs, 119, 120
- tda_mevs(), 110–113, 116, 120, 122–127, 134–139, 141, 142, 148, 149, 151, 153
- tda_mginv, 148
- tda_mginv(tda_minvs), 125
- tda_midf, 121, 122–124
- tda_midf(), 110–113, 116, 120, 121, 123–127, 134–139, 141, 142, 148,

- [149, 151, 153](#)
 tda_midf1, [122, 124](#)
 tda_midf1(), [110–113, 116, 120–127, 134–139, 141, 142, 148, 149, 151, 153](#)
 tda_midf2, [123, 124](#)
 tda_midf2(), [110–113, 116, 120–127, 134–139, 141, 142, 148, 149, 151, 153](#)
 tda_midf3, [124](#)
 tda_midf3(), [110–113, 116, 120–123, 125–127, 134–139, 141, 142, 148, 149, 151, 153](#)
 tda_minimize(tda_integrate), [86](#)
 tda_minvd, [149](#)
 tda_minvd(tda_minvs), [125](#)
 tda_minvs, [125](#)
 tda_minvs(), [110–113, 116, 120–124, 126, 127, 134–139, 141, 142, 148, 149, 151, 153](#)
 tda_mivec(tda_mcvvec), [114](#)
 tda_mkmet, [126](#)
 tda_mkmet(), [110–113, 116, 120–125, 127, 134–139, 141, 142, 148, 149, 151, 153](#)
 tda_mkp, [127](#)
 tda_mkp(), [110–113, 116, 120–126, 134–139, 141, 142, 148, 149, 151, 153](#)
 tda_mlp, [106, 127](#)
 tda_mlp(), [15, 130, 133, 143, 145](#)
 tda_mlp1(tda_mlp), [127](#)
 tda_mlpi, [129](#)
 tda_mlpi(), [15, 129, 133, 143, 145](#)
 tda_mlrc_design, [130](#)
 tda_mlrc_design(), [50, 55, 70, 96, 103, 146, 156, 194, 266](#)
 tda_mls, [131](#)
 tda_mls(), [15, 129, 130, 143, 145](#)
 tda_mlse(tda_mls), [131](#)
 tda_mlsei(tda_mls), [131](#)
 tda_mlsei1, [144](#)
 tda_mlsei1(tda_mls), [131](#)
 tda_mlsi(tda_mls), [131](#)
 tda_mmul, [133](#)
 tda_mmul(), [110–113, 116, 120–127, 135–139, 141, 142, 148, 149, 151, 153](#)
 tda_mnc, [134](#)
 tda_mnc(), [110–113, 116, 120–127, 134, 136–139, 141, 142, 148, 149, 151, 153](#)
 tda_mncol(tda_mnrow), [135](#)
 tda_mnlis(tda_mls), [131](#)
 tda_mnorm(tda_mnrow), [135](#)
 tda_mnorm1(tda_mnrow), [135](#)
 tda_mnorm2(tda_mnrow), [135](#)
 tda_mnrow, [135](#)
 tda_mnrow(), [110–113, 116, 120–127, 134–139, 141, 142, 148, 149, 151, 153](#)
 tda_mnum(tda_mcross), [113](#)
 tda_mparse(tda_expressions), [46](#)
 tda_mpbl, [112, 136](#)
 tda_mpbl(), [110–113, 116, 120–127, 134–139, 141, 142, 148, 149, 151, 153](#)
 tda_mpbu(tda_mpbl), [136](#)
 tda_mpcol(tda_msrow), [150](#)
 tda_mpfir, [137](#)
 tda_mpfir(), [110–113, 116, 120–127, 134–136, 138, 139, 141, 142, 148, 149, 151, 153](#)
 tda_mpinv, [138](#)
 tda_mpinv(), [110–113, 116, 120–127, 134–137, 139, 141, 142, 148, 149, 151, 153](#)
 tda_mpit, [139](#)
 tda_mpit(), [110–113, 116, 120–127, 134–138, 141, 142, 148, 149, 151, 153](#)
 tda_mpit1(tda_mpit), [139](#)
 tda_mple, [105, 140](#)
 tda_mple(), [110–113, 116, 120–127, 134–139, 142, 148, 149, 151, 153](#)
 tda_mproc, [141](#)
 tda_mprorow(tda_msrow), [150](#)
 tda_mpsym(tda_msrow), [150](#)
 tda_mpz, [112, 142](#)
 tda_mpz(), [110–113, 116, 120–127, 134–139, 141, 148, 149, 151, 153](#)
 tda_mqap, [143](#)
 tda_mqap(), [15, 129, 130, 133, 145](#)
 tda_mqp, [144](#)
 tda_mqp(), [15, 129, 130, 133, 143](#)
 tda_mqpb(tda_mqp), [144](#)
 tda_mqpc(tda_mqp), [144](#)

- `tda_mrank (tda_msort)`, 148
- `tda_mreg`, 145
- `tda_mreg()`, 50, 55, 70, 96, 103, 131, 156, 194, 266
- `tda_mrsum`, 146
- `tda_mrsum()`, 105, 109, 114, 149, 150, 152
- `tda_mrvec (tda_mcvec)`, 114
- `tda_mscal1`, 147
- `tda_mscal1()`, 110–113, 116, 120–127, 134–139, 141, 142, 149, 151, 153
- `tda_mscol (tda_msrow)`, 150
- `tda_msort`, 148
- `tda_msort()`, 105, 109, 114, 147, 150, 152
- `tda_msort1 (tda_msort)`, 148
- `tda_msqrd`, 149
- `tda_msqrd()`, 110–113, 116, 120–127, 134–139, 141, 142, 148, 151, 153
- `tda_msqrtd (tda_msqrtd)`, 149
- `tda_msrow`, 150
- `tda_msrow()`, 105, 109, 114, 147, 149, 152
- `tda_mstand (tda_mcent)`, 111
- `tda_msvd`, 151
- `tda_msvd()`, 110–113, 116, 120–127, 134–139, 141, 142, 148, 149, 153
- `tda_msvd1 (tda_msvd)`, 151
- `tda_mtrace (tda_mnrow)`, 135
- `tda_mtransp (tda_mmul)`, 133
- `tda_mtrim`, 152
- `tda_mtrim()`, 105, 109, 114, 147, 149, 150
- `tda_mwvec`, 153
- `tda_mwvec()`, 110–113, 116, 120–127, 134–139, 141, 142, 148, 149, 151
- `tda_mwvec1 (tda_mwvec)`, 153
- `tda_ndvar`, 154
- `tda_niset`, 154
- `tda_nlreg`, 55
- `tda_nlreg()`, 50, 55, 70, 96, 103, 131, 146, 156, 194, 266
- `tda_npreg`, 155, 165
- `tda_npreg()`, 50, 55, 70, 96, 103, 131, 146, 194, 266
- `tda_nvar (tda_write_data)`, 260
- `tda_options`, 156
- `tda_output`, 157
- `tda_output()`, 18, 76, 209, 221, 262
- `tda_output_audit`, 158
- `tda_payload`, 159
- `tda_pccyc`, 159
- `tda_pdatd`, 160
- `tda_pdatd()`, 21, 24, 29, 34, 117
- `tda_pdatr`, 161
- `tda_pl`, 162, 170, 172
- `tda_pl()`, 10, 19, 167–169, 172, 174, 176, 178, 180, 186, 189, 202
- `tda_pl_arc`, 166
- `tda_pl_arc()`, 10, 19, 165, 168, 169, 172, 174, 176, 178, 180, 186, 189, 202
- `tda_pl_axes`, 168
- `tda_pl_axes (tda_pl)`, 162
- `tda_pl_axis`, 167
- `tda_pl_axis()`, 10, 19, 165, 167, 169, 172, 174, 176, 178, 180, 186, 189, 202
- `tda_pl_circle (tda_pl)`, 162
- `tda_pl_contour (tda_pl_hist)`, 170
- `tda_pl_curve (tda_pl_hist)`, 170
- `tda_pl_curve3 (tda_ps3)`, 187
- `tda_pl_density (tda_pl_hist)`, 170
- `tda_pl_ellipse (tda_pl_hist)`, 170
- `tda_pl_frame (tda_pl)`, 162
- `tda_pl_function (tda_pl)`, 162
- `tda_pl_graph`, 7, 8, 169
- `tda_pl_graph()`, 10, 19, 165, 167, 168, 172, 174, 176, 178, 180, 186, 189, 202
- `tda_pl_grid (tda_pl)`, 162
- `tda_pl_grid_lines (tda_pl_hist)`, 170
- `tda_pl_hist`, 10, 19, 165, 167–169, 170, 174, 176, 178, 180, 186, 189, 202
- `tda_pl_histogram (tda_pl_hist)`, 170
- `tda_pl_hull (tda_pl_regression)`, 175
- `tda_pl_labels`, 225
- `tda_pl_labels (tda_pl_hist)`, 170
- `tda_pl_lines`, 166, 176, 225
- `tda_pl_lines (tda_pl)`, 162
- `tda_pl_lines3`, 225
- `tda_pl_lines3 (tda_ps3)`, 187
- `tda_pl_panel`, 173, 185
- `tda_pl_panel()`, 10, 19, 165, 167–169, 172, 176, 178, 180, 186, 189, 202
- `tda_pl_points (tda_pl)`, 162
- `tda_pl_points3`, 225
- `tda_pl_points3 (tda_ps3)`, 187
- `tda_pl_polyline (tda_pl)`, 162
- `tda_pl_rect (tda_pl)`, 162
- `tda_pl_regression`, 175
- `tda_pl_regression()`, 10, 19, 165, 167–169, 172, 174, 178, 180, 186, 189, 202

- tda_pl_scatter, [175, 176, 177](#)
- tda_pl_scatter(), [10, 19, 165, 167–169, 172, 174, 176, 180, 186, 189, 202](#)
- tda_pl_smooth(tda_pl), [162](#)
- tda_pl_surface3(tda_ps3), [187](#)
- tda_pl_text(tda_pl), [162](#)
- tda_pl_text3, [225](#)
- tda_pl_text3(tda_ps3), [187](#)
- tda_plcirc3, [178](#)
- tda_ple, [30, 48, 140, 255](#)
- tda_ple(tda_dple), [36](#)
- tda_plglob3, [179](#)
- tda_plot_ps, [180](#)
- tda_plot_ps(), [10, 19, 165, 167–169, 172, 174, 176, 178, 186, 189, 202](#)
- tda_plotcm, [181](#)
- tda_plotr, [181](#)
- tda_plotsp, [182](#)
- tda_plsurf3, [183](#)
- tda_polygons, [184, 223](#)
- tda_polygons(), [11, 107, 201, 203, 205, 222, 224, 228, 249](#)
- tda_ps, [168, 169, 173–175, 185, 225](#)
- tda_ps(), [10, 19, 165, 167–169, 172, 174, 176, 178, 180, 189, 202](#)
- tda_ps3, [187](#)
- tda_ps3(), [10, 19, 165, 167–169, 172, 174, 176, 178, 180, 186, 202](#)
- tda_ps_file, [22](#)
- tda_ps_file(tda_pl), [162](#)
- tda_ps_objects, [186](#)
- tda_ptree, [191](#)
- tda_ptree(), [8, 35, 60, 65, 74, 97, 259](#)
- tda_qreg, [6, 30, 48, 49, 51, 238](#)
- tda_qreg(TDA_QRMODELS), [192](#)
- TDA_QRMODELS, [50, 55, 70, 96, 103, 131, 146, 156, 192, 266](#)
- tda_quant, [195](#)
- tda_quant(), [17, 28, 40, 57, 84, 85, 99, 199, 231, 254](#)
- tda_range(tda_integrate), [86](#)
- tda_rank_orders, [196](#)
- tda_rap, [197](#)
- tda_rate, [9, 25, 30, 57–59, 198, 238, 250](#)
- tda_rates, [198](#)
- tda_rates(), [25, 27, 38, 251, 255, 257](#)
- tda_rcorr, [198](#)
- tda_rcorr(), [17, 28, 40, 57, 84, 85, 99, 196, 231, 254](#)
- tda_rcsv, [199, 217](#)
- tda_rd, [200](#)
- tda_rdn(tda_rd), [200](#)
- tda_read_blocks(tda_read_table), [209](#)
- tda_read_dbf, [201](#)
- tda_read_dbf(), [11, 107, 184, 203, 205, 222, 224, 228, 249](#)
- tda_read_dcw(tda_read_spatial), [203](#)
- tda_read_e00(tda_read_spatial), [203](#)
- tda_read_gshhs, [157](#)
- tda_read_gshhs(tda_read_spatial), [203](#)
- tda_read_ps, [19, 180, 186, 187, 202](#)
- tda_read_ps(), [10, 19, 165, 167–169, 172, 174, 176, 178, 180, 186, 189](#)
- tda_read_shapefile, [201, 203, 203, 223](#)
- tda_read_shapefile(), [11, 107, 184, 201, 205, 222, 224, 228, 249](#)
- tda_read_spatial, [11, 107, 184, 201, 203, 203, 222, 224, 228, 249](#)
- tda_read_spss, [206, 208, 264](#)
- tda_read_spss(), [212, 253](#)
- tda_read_stata, [207](#)
- tda_read_sys, [106, 208, 220, 263, 264](#)
- tda_read_table, [43, 105, 209](#)
- tda_read_table(), [18, 76, 158, 221, 262](#)
- tda_read_ucinet, [210](#)
- tda_read_xls, [211](#)
- tda_read_xls(), [207, 253](#)
- tda_recode, [212](#)
- tda_repsel, [213](#)
- tda_rfit, [214](#)
- tda_rmod, [215](#)
- tda_rng, [215](#)
- tda_rnorm(tda_rng), [215](#)
- tda_rod, [217](#)
- tda_rplz, [217](#)
- tda_rrdat, [218](#)
- tda_rsys, [219](#)
- tda_run, [12–15, 17, 18, 20–24, 29, 31, 34–37, 41, 42, 47, 49, 52, 55, 58, 60, 67, 68, 70–73, 75, 79, 80, 82, 83, 85, 90, 92, 94, 96, 97, 102, 106, 115, 117–119, 128, 130, 132, 141, 143, 144, 146, 154, 155, 157–161, 178, 179, 181–183, 186, 191, 193, 196, 197, 199, 201, 206–208, 210, 212–215, 217–219, 220, 221, 222, 230, 232,](#)

- 235, 237, 240, 242, 244–246,
251–253, 256–258, 260, 264, 265
- `tda_run()`, 18, 76, 158, 209, 262
- `tda_run_cf` (`tda_run`), 220
- `tda_runif`, 200
- `tda_runif` (`tda_rng`), 215
- `tda_scla`, 221
- `tda_sd`, 222
- `tda_sd()`, 11, 107, 184, 201, 203, 205, 224,
228, 249
- `tda_sd_analyses`, 11, 107, 184, 201, 203,
205, 222, 223, 228, 249
- `tda_sd_clip` (`tda_sd_analyses`), 223
- `tda_sd_data` (`tda_sd_analyses`), 223
- `tda_sd_enclosing` (`tda_sd_analyses`), 223
- `tda_sd_info`, 228
- `tda_sd_info` (`tda_sd_analyses`), 223
- `tda_sd_intersect` (`tda_sd_analyses`), 223
- `tda_sd_lines` (`tda_sd_analyses`), 223
- `tda_sd_neighbours` (`tda_sd_analyses`), 223
- `tda_sd_pl3` (`tda_sd_ps3`), 225
- `tda_sd_pl3_lines` (`tda_sd_ps3`), 225
- `tda_sd_pl3_points` (`tda_sd_ps3`), 225
- `tda_sd_pl3_polygons` (`tda_sd_ps3`), 225
- `tda_sd_plot3`, 11
- `tda_sd_plot3` (`tda_sd_ps3`), 225
- `tda_sd_plot3_lines` (`tda_sd_ps3`), 225
- `tda_sd_plot3_polygons` (`tda_sd_ps3`), 225
- `tda_sd_points` (`tda_sd_analyses`), 223
- `tda_sd_ps3`, 225
- `tda_sd_ps3()`, 11, 107, 184, 201, 203, 205,
222, 224, 249
- `tda_sd_relations` (`tda_sd_analyses`), 223
- `tda_sd_select` (`tda_sd_analyses`), 223
- `tda_sd_voronoi` (`tda_sd_analyses`), 223
- `tda_sddf`, 229
- `tda_sddf()`, 16, 79, 80, 82, 83, 85, 92, 95
- `tda_segr`, 231
- `tda_segr()`, 17, 28, 40, 57, 84, 85, 99, 196,
199, 254
- `tda_seq_info`, 232
- `tda_seq_info()`, 46, 53, 59, 235, 238, 241,
244, 245
- `tda_seqen` (`tda_seqgc`), 233
- `tda_seqv` (`tda_seqmd`), 238
- `tda_seqv` (`tda_seqmd`), 238
- `tda_seqgc`, 233
- `tda_seqgc()`, 46, 53, 59, 233, 238, 241, 244,
245
- `tda_seqlg` (`tda_seqgc`), 233
- `tda_seqm`, 20, 232, 234, 236, 239, 242, 243,
245
- `tda_seqm()`, 46, 53, 59, 233, 235, 241, 244,
245
- `tda_seqmd`, 238
- `tda_seqmd()`, 46, 53, 59, 233, 235, 238, 244,
245
- `tda_seqpd`, 241
- `tda_seqpe`, 243
- `tda_seqpe()`, 46, 53, 59, 233, 235, 238, 241,
245
- `tda_seqpm`, 244
- `tda_seqpm()`, 46, 53, 59, 233, 235, 238, 241,
244
- `tda_seqs` (`tda_seqgc`), 233
- `tda_seqs` (`tda_seqgc`), 233
- `tda_sga`, 246
- `tda_sma`, 246
- `tda_sma()`, 87, 88, 91, 108, 248, 250
- `tda_smd`, 247
- `tda_smd()`, 87, 88, 91, 108, 247, 250
- `tda_spatial`, 11, 107, 184, 203, 222, 223,
227, 248
- `tda_spatial()`, 11, 107, 184, 201, 203, 205,
222, 224, 228
- `tda_spl`, 249
- `tda_spl()`, 87, 88, 91, 108, 247, 248
- `tda_split`, 250
- `tda_split()`, 25, 27, 38, 198, 255, 257
- `tda_spm`, 252
- `tda_state_dist` (`tda_episodes`), 41
- `tda_strict` (`tda_control`), 26
- `tda_strings`, 253, 262
- `tda_strings()`, 207, 212
- `tda_subm`, 254
- `tda_subm()`, 17, 28, 40, 57, 84, 85, 99, 196,
199, 231
- `tda_survivor`, 198, 255
- `tda_survivor()`, 25, 27, 38, 198, 251, 257
- `tda_time` (`tda_help`), 76
- `tda_tnet`, 256
- `tda_transitions`, 256
- `tda_transitions()`, 25, 27, 38, 198, 251, 255
- `tda_ttriang`, 257
- `tda_ucl`, 258
- `tda_ucl()`, 8, 35, 60, 65, 74, 97, 191

tda_unf, [259](#)
tda_write_data, [253](#), [260](#)
tda_write_data(), [18](#), [76](#), [158](#), [209](#), [221](#)
tda_write_foreign, [263](#)
tda_write_spss (tda_write_foreign), [263](#)
tda_write_stata, [207](#), [208](#)
tda_write_stata (tda_write_foreign), [263](#)
tda_write_sys, [208](#)
tda_write_sys (tda_write_foreign), [263](#)
tda_zreg, [264](#)
tda_zreg(), [50](#), [55](#), [70](#), [96](#), [103](#), [131](#), [146](#),
[156](#), [194](#)
tda_zreg1 (tda_zreg), [264](#)
terms, [99](#)

unzip, [266](#), [267](#)
unzoo, [266](#)

vcov.tda_fit(), [25](#), [27](#), [38](#), [198](#), [251](#), [255](#),
[257](#)
vcov.tda_loglin (tda_loglin), [98](#)

zip, [266](#)
zoo (unzoo), [266](#)