

Package ‘tf’

July 14, 2026

Title S3 Classes and Methods for Tidy Functional Data

Version 0.5.0

Description Provides S3 vector types for functional data represented on grids, in spline bases, or via functional principal components. Supports arithmetic and summary methods, plotting, derivation, integration, smoothing, registration, and data import/export for these functional vectors. Includes data-wrangling tools for re-evaluation, subsetting, sub-assignment, zooming into sub-domains, and extracting functional features such as minima, maxima, and their locations. Enables joint analysis of functional and scalar variables by integrating functional vectors into standard data frames.

License AGPL (>= 3)

URL <https://tidyfun.github.io/tf/>, <https://github.com/tidyfun/tf/>

BugReports <https://github.com/tidyfun/tf/issues>

Depends R (>= 4.1)

Imports checkmate, cli, methods, mgcv, purrr (>= 1.0.0), rlang, stats, vctrs (>= 0.2.4), zoo

Suggests covr, dplyr, fda, fdasrvf (>= 2.4.0), pillar, refund, testthat (>= 3.0.0), tibble, tidyr, withr

Config/testthat/edition 3

Encoding UTF-8

LazyData true

Collate 'tfb-class.R' 'tfd-class.R' 'tfd-mv.R' 'tfb-mv.R' 'vctrs-mv.R' 'accessors-mv.R' 'approx.R' 'assertions.R' 'bibentries.R' 'brackets-mv.R' 'brackets.R' 'calculus-mv.R' 'calculus.R' 'convert-construct-utils.R' 'convert-mv.R' 'convert.R' 'data.R' 'depth.R' 'evaluate.R' 'fda-connectors.R' 'fwise.R' 'geometry-mv.R' 'globals.R' 'graphics.R' 'interpolate.R' 'landmarks.R' 'ops.R' 'math.R' 'methods.R' 'mv-stubs.R' 'ops-mv.R' 'plot-mv.R' 'print-format-mv.R' 'print-format.R' 'rebase.R' 'register-cc.R' 'register-mv.R' 'register-utils.R' 'register.R' 'registration-class.R' 'rng.R' 'smooth.R'

'soft-impute-svd.R' 'split-combine.R' 'summarize.R'
 'tf-package.R' 'tfb-fpc.R' 'tfb-spline.R' 'tf-s4.R'
 'tfb-fpc-utils.R' 'tfb-mfpc.R' 'tfb-spline-utils.R' 'utils.R'
 'vctrs-cast.R' 'vctrs-ptype2.R' 'where.R' 'zoom.R' 'zzz.R'

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Fabian Scheipl [aut, cre, cph] (ORCID:

<<https://orcid.org/0000-0001-8172-3603>>),

Jeff Goldsmith [aut],

Maximilian Mücke [aut] (ORCID: <<https://orcid.org/0009-0000-9432-9795>>),

Julia Wrobel [ctb] (ORCID: <<https://orcid.org/0000-0001-6783-1421>>),

Sebastian Fischer [ctb] (ORCID:

<<https://orcid.org/0000-0002-9609-3197>>)

Maintainer Fabian Scheipl <fabian.scheipl@gmail.com>

Repository CRAN

Date/Publication 2026-07-14 10:40:02 UTC

Contents

as.data.frame.tf	3
converters-mv	5
ensure_list	6
fivenum	7
fpc_wsvd	8
functionwise	9
gait	12
growth	13
in_range	14
median.tf_mv	14
pinch	15
plot.tf	16
plot.tf_mv	18
prep_plotting_arg	19
print.tf	19
tfb	21
tfb_fpc	22
tfb_mfpc	25
tfb_mv	27
tfb_spline	29
tfd	33
tfd_mv	36
tfgroupgenerics	38
tfsummaries	40
tf_align	42
tf_approx_linear	43
tf_arclength	45

tf_arg	46
tf_depth	49
tf_derive	51
tf_estimate_warps	53
tf_evaluate	55
tf_geom	57
tf_integrate	59
tf_interpolate	60
tf_invert	62
tf_jiggle	62
tf_landmarks_extrema	63
tf_minmax	65
tf_mv_methods	66
tf_order	67
tf_rebase	69
tf_register	70
tf_register_shape	73
tf_registration	74
tf_rgp	77
tf_smooth	78
tf_split	80
tf_warp	81
tf_where	83
tf_zoom	85
unique_id	86
vctrs	87
[.tf_mv	90

Index 93

as.data.frame.tf	<i>Convert functional data back to tabular data formats</i>
------------------	---

Description

Various converters to turn tfb- or tfd-vectors into data.frames or matrices, or even an actual R function.

Usage

```
## S3 method for class 'tf'
as.data.frame(x, row.names = NULL, optional = FALSE, unnest = FALSE, ...)

## S3 method for class 'tf'
as.matrix(x, arg, interpolate = FALSE, ...)

## S3 method for class 'tf'
as.function(x, ...)
```

Arguments

<code>x</code>	a tf object.
<code>row.names</code>	NULL or a character vector giving the row names for the data frame. Missing values are not allowed.
<code>optional</code>	not used.
<code>unnest</code>	if TRUE, the function will return a data.frame with the evaluated functions.
<code>...</code>	additional arguments to be passed to or from methods.
<code>arg</code>	a vector of argument values / evaluation points for x. Defaults to <code>tf_arg(x)</code> (so for x on irregular grids, this will be the union of all observed arg-values by default).
<code>interpolate</code>	should functions be evaluated (i.e., inter-/extrapolated) for values in <code>arg</code> for which no original data is available? Only relevant for the raw data class <code>tfd</code> , for which it defaults to FALSE. Basis-represented functional data <code>tfb</code> are always "interpolated".

Value

for `as.data.frame.tf`: if `unnest` is FALSE (default), a one-column data.frame with a `tf`-column containing `x`. if `unnest` is TRUE, a 3-column data frame with columns `id` (containing (unique) names of `x` or a numeric identifier if `x` is unnamed), `arg`, and `value`, with each row containing one function evaluation at the original `arg`-values.

for `as.matrix.tf`: a matrix with one row per function and one column per `arg`.

for `as.function.tf`: an R function with argument `arg` that evaluates `x` on `arg` and returns the list of function values

See Also

Other tidyfun converters: [converters-mv](#)

Examples

```
f <- tfd(sin(seq(0, 2 * pi, length.out = 11)), arg = seq(0, 1, length.out = 11))
as.data.frame(f)
as.data.frame(f, unnest = TRUE)
as.matrix(f)
fun <- as.function(f)
fun(c(0, 0.5, 1))
```

converters-mv

Coerce a tf_mv to a matrix or data.frame

Description

`as.matrix.tf_mv` returns a **3-d** array `[curve, arg, component]` – the natural shape for a vector-valued evaluation. This is deliberately different from `as.matrix.tf` (2-d, `[curve, arg]`); see @seealso.

Usage

```
## S3 method for class 'tf_mv'
as.matrix(x, arg, interpolate = FALSE, ...)

## S3 method for class 'tf_mv'
as.data.frame(
  x,
  row.names = NULL,
  optional = FALSE,
  unnest = FALSE,
  long = TRUE,
  arg = NULL,
  interpolate = TRUE,
  grids = c("union", "component"),
  ...
)
```

Arguments

<code>x</code>	a <code>tf_mv</code> object.
<code>arg</code>	optional evaluation grid (numeric vector or per-curve list). When <code>NULL</code> (default for <code>as.data.frame.tf_mv</code> ; equivalent to "missing" for <code>as.matrix.tf_mv</code>), the per-curve union of all components' native argument grids is used.
<code>interpolate</code>	forwarded to the underlying <code>tf</code> evaluation. <code>tfb</code> components are always interpolated.
<code>...</code>	passed through.
<code>row.names, optional</code>	standard <code>as.data.frame</code> plumbing.
<code>unnest</code>	if <code>TRUE</code> , return an evaluated <code>data.frame</code> (see <code>long</code>); if <code>FALSE</code> (default), a one-column <code>data.frame</code> wrapping <code>x</code> .
<code>long</code>	when <code>unnest = TRUE</code> , controls the schema. <code>long = TRUE</code> (default) returns a 4-column <code>data.frame</code> (<code>id, arg, component, value</code>) – the multivariate analogue of the univariate (<code>id, arg, value</code>) contract, with <code>component</code> a factor over <code>attr(x, "comp_names")</code> . <code>long = FALSE</code> returns the wide (<code>id, arg, comp1, ..., compd</code>) schema.

`grids` when `unnest = TRUE`, controls *where* components are evaluated when they live on different argument grids (for shared grids both settings agree). "union" (default) evaluates every component on each curve's union grid, so components get (interpolated) values at the *other* components' arg values inside their observed range. "component" evaluates each component strictly on its **own** grid (or on `arg`, if supplied): no values are fabricated at args a component was not observed at – in the long schema such rows are simply absent, in the wide schema the other components' columns are NA there. Use "union" for paired evaluations (e.g. trajectory plots), "component" for faithful tabular exports of the observed data.

Details

`as.data.frame.tf_mv` returns either a single-column wrapping data.frame (`unnest = FALSE`, for storing a `tf_mv` in a tibble column) or an evaluated long/wide data.frame (`unnest = TRUE`).

Value

a 3-d array (`as.matrix.tf_mv`) or a data.frame (`as.data.frame.tf_mv`).

See Also

[as.matrix.tf\(\)](#) (2-d sibling), [as.data.frame.tf\(\)](#) (univariate contract), [tf_evaluate\(\)](#).

Other tidyfun converters: [as.data.frame.tf\(\)](#)

Examples

```
arg <- seq(0, 1, length.out = 11)
xf <- tfd(t(sapply(1:3, function(i) sin(2 * pi * arg + i))), arg = arg)
yf <- tfd(t(sapply(1:3, function(i) cos(2 * pi * arg + i))), arg = arg)
mv <- tfd_mv(list(x = xf, y = yf))
dim(as.matrix(mv))
head(as.data.frame(mv, unnest = TRUE))
```

`ensure_list`

Wrap a non-list object in a list

Description

Returns `x` unchanged if it is already a list, otherwise wraps it in a one-element list. Used internally to normalize `arg` inputs that may be either a single numeric vector or a list of per-curve vectors.

Usage

```
ensure_list(x)
```

Arguments

`x` any input.

Value

`x` if it is a list, otherwise `list(x)`.

See Also

Other tidyfun utility functions: [in_range\(\)](#), [tf_arg\(\)](#), [tf_zoom\(\)](#), [unique_id\(\)](#)

Examples

```
ensure_list(1:3)
ensure_list(list(1:3, 4:6))
```

fivenum	<i>Tukey's Five Number Summary for tf vectors</i>
---------	---

Description

Computes a depth-based five number summary for functional data: the observations with minimum, lower-hinge, median, upper-hinge, and maximum depth values.

Usage

```
fivenum(x, na.rm = FALSE, ...)

## Default S3 method:
fivenum(x, na.rm = FALSE, ...)

## S3 method for class 'tf'
fivenum(x, na.rm = FALSE, depth = "MHI", ...)

## S3 method for class 'tf_mv'
fivenum(x, na.rm = FALSE, depth = "MBD", ...)
```

Arguments

<code>x</code>	a tf vector (or numeric for the default method).
<code>na.rm</code>	logical; if TRUE, NA observations are removed first.
<code>...</code>	passed to tf_depth() .
<code>depth</code>	depth method for ordering. See tf_depth() . Defaults to "MHI" for an up-down ordering.

Value

`fivenum.tf`: a named tf vector of length 5.
`fivenum.default`: see [stats::fivenum\(\)](#).

See Also

Other tidyfun summary functions: [functionwise](#), [median.tf_mv\(\)](#), [tfsummaries](#)

Examples

```
set.seed(1)
f <- tf_rgp(7)
fivenum(f)
```

fpc_wsvd

Eigenfunctions via weighted, regularized SVD

Description

Compute (truncated) orthonormal eigenfunctions and scores for (partially missing) data on a common (potentially non-equidistant) grid.

Usage

```
fpc_wsvd(data, arg, pve = 0.995)

## S3 method for class 'matrix'
fpc_wsvd(data, arg, pve = 0.995)

## S3 method for class 'data.frame'
fpc_wsvd(data, arg, pve = 0.995)
```

Arguments

data	numeric matrix of function evaluations (each row is one curve, no NAs).
arg	numeric vector of argument values.
pve	percentage of variance explained.

Details

Performs a weighted SVD with trapezoidal quadrature weights s.t. returned vectors represent (evaluations of) orthonormal *eigenfunctions* $\phi_j(t)$, not *eigenvectors* $\phi_j = (\phi_j(t_1), \dots, \phi_j(t_n))$, specifically:

$\int_T \phi_j(t)^2 dt \approx \sum_i \Delta_i \phi_j(t_i)^2 = 1$ given quadrature weights Δ_i , not $\phi_j' \phi_j = \sum_i \phi_j(t_i)^2 = 1$;

$\int_T \phi_j(t) \phi_k(t) dt = 0$ not $\phi_j' \phi_k = \sum_i \phi_j(t_i) \phi_k(t_i) = 0$.

For incomplete data, this uses a soft-impute iterative-SVD scheme (see references). Note that this will not work well for data on a common grid if more than a few percent of data points are missing, and it breaks down completely for truly irregular data with no/few common timepoints, even if observed very densely. For such data, either re-evaluate on a common grid first or use more advanced FPCA approaches like `refund::fPCA_sc()`, see last example for [tfb_fpc\(\)](#)

Value

a list with entries

- `mu` estimated mean function (numeric vector)
- `efunctions` estimated FPCs (numeric matrix, columns represent FPCs)
- `scores` estimated FPC scores (one row per observed curve)
- `npc` how many FPCs were returned for the given pve (integer)
- `scoring_function` a function that returns FPC scores for new data and given eigenfunctions, see `tf:::fpc_wsvd_scores` for an example.

Author(s)

Fabian Scheipl

References

the soft-impute SVD algorithm for incomplete data is described in Mazumder, Rahul, Hastie, Trevor, Tibshirani, Robert (2010). “Spectral Regularization Algorithms for Learning Large Incomplete Matrices.” *The Journal of Machine Learning Research*, **11**, 2287–2322.

See Also

Other `tfb`-class: [tfb\(\)](#), [tfb_fpc\(\)](#), [tfb_spline\(\)](#)

Other `tfb_fpc`-class: [tfb_fpc\(\)](#), [tfb_mfpc\(\)](#)

Examples

```
arg <- seq(0, 1, length.out = 41)
data <- t(sapply(seq(0, 2 * pi, length.out = 10),
  function(p) sin(2 * pi * arg + p)))
fpc <- fpc_wsvd(data, arg = arg, pve = 0.99)
fpc$npc
```

functionwise

Summarize each `tf` in a vector (function-wise)

Description

These functions extract (user-specified) **function-wise** summary statistics from every entry in a `tf`-vector. To summarize a vector of functions at each argument value, see `?tfsummaries`. Note that most of these will tend to yield lots of NAs for irregular `tf`d unless you set a [tf_evaluator\(\)](#)-function that does inter- and extrapolation for them beforehand.

Usage

```

tf_fwise(x, .f, arg = tf_arg(x), ...)

tf_fmax(x, arg = tf_arg(x), na.rm = FALSE)

## S3 method for class 'tf_mv'
tf_fmax(x, arg = tf_arg(x), na.rm = FALSE)

tf_fmin(x, arg = tf_arg(x), na.rm = FALSE)

## S3 method for class 'tf_mv'
tf_fmin(x, arg = tf_arg(x), na.rm = FALSE)

tf_fmedian(x, arg = tf_arg(x), na.rm = FALSE)

## S3 method for class 'tf_mv'
tf_fmedian(x, arg = tf_arg(x), na.rm = FALSE)

tf_frange(x, arg = tf_arg(x), na.rm = FALSE, finite = FALSE)

tf_fmean(x, arg = tf_arg(x))

## S3 method for class 'tf_mv'
tf_fmean(x, arg = tf_arg(x))

tf_fvar(x, arg = tf_arg(x))

## S3 method for class 'tf_mv'
tf_fvar(x, arg = tf_arg(x))

tf_fsd(x, arg = tf_arg(x))

## S3 method for class 'tf_mv'
tf_fsd(x, arg = tf_arg(x))

tf_crosscov(x, y, arg = tf_arg(x))

tf_crosscor(x, y, arg = tf_arg(x))

```

Arguments

<code>x</code>	a <code>tf</code> object.
<code>.f</code>	a function or formula that is applied to each entry of <code>x</code> , see purrr::as_mapper() and details.
<code>arg</code>	defaults to standard argument values of <code>x</code> .
<code>...</code>	additional arguments for purrr::as_mapper() .
<code>na.rm</code>	a logical (TRUE or FALSE) indicating whether missing values should be removed.

`finite` logical, indicating if all non-finite elements should be omitted.
`y` a `tf` object.

Details

`tf_fwise` turns `x` into a list of data.frames with columns `arg` and values internally, so the function/formula in `.f` gets a data.frame `.x` with these columns, see examples below or source code for [tf_fmin\(\)](#), [tf_fmax\(\)](#), etc.

Value

a list (or vector) of the same length as `x` with the respective summaries.

Functions

- `tf_fwise()`: User-specified function-wise summary statistics
- `tf_fmax()`: maximal value of each function
- `tf_fmax(tf_mv)`: component-wise maxima of each vector-valued function
- `tf_fmin()`: minimal value of each function
- `tf_fmin(tf_mv)`: component-wise minima of each vector-valued function
- `tf_fmedian()`: median value of each function
- `tf_fmedian(tf_mv)`: component-wise medians of each vector-valued function
- `tf_frange()`: range of values of each function
- `tf_fmean()`: mean of each function: $\frac{1}{|T|} \int_T x_i(t) dt$
- `tf_fmean(tf_mv)`: component-wise means of each vector-valued function
- `tf_fvar()`: variance of each function: $\frac{1}{|T|} \int_T (x_i(t) - \bar{x}(t))^2 dt$
- `tf_fvar(tf_mv)`: component-wise variances of each vector-valued function
- `tf_fsd()`: standard deviation of each function: $\sqrt{\frac{1}{|T|} \int_T (x_i(t) - \bar{x}(t))^2 dt}$
- `tf_fsd(tf_mv)`: component-wise standard deviations of each vector-valued function
- `tf_crosscov()`: cross-covariances between two functional vectors: $\frac{1}{|T|} \int_T (x_i(t) - \bar{x}(t))(y_i(t) - \bar{y}(t)) dt$
- `tf_crosscor()`: cross-correlation between two functional vectors: `tf_crosscov(x, y) / (tf_fsd(x) * tf_fsd(y))`

See Also

Other tidyfun summary functions: [fivenum\(\)](#), [median.tf_mv\(\)](#), [tfsummaries](#)

Examples

```
x <- tf_rgp(3)
layout(t(1:3))
plot(x, col = 1:3)
# each function's values to [0,1]:
x_clamp <- (x - tf_fmin(x)) / (tf_fmax(x) - tf_fmin(x))
plot(x_clamp, col = 1:3)
# standardize each function to have mean / integral 0 and sd 1:
x_std <- (x - tf_fmean(x)) / tf_fsd(x)
tf_fvar(x_std) == c(1, 1, 1)
plot(x_std, col = 1:3)
# Custom functions:
# 80%tiles of each function's values:
tf_fwise(x, \(x) quantile(.x$value, 0.8)) |> unlist()
# minimal value of each function for t > 0.5
tf_fwise(x, \(x) min(.x$value[.x$arg > 0.5])) |> unlist()

tf_crosscor(x, -x)
tf_crosscov(x, x) == tf_fvar(x)
```

gait

Hip and knee angle while walking data

Description

Hip and knee angle measurements in degrees through a 20-point movement cycle for 39 children. The data represents the angular positions of hip and knee joints during normal walking gait, captured at evenly spaced time points throughout the gait cycle.

Usage

```
gait
```

Format

A data frame with 39 rows and 3 variables:

subject_id subject identifier

knee_angle a tfd column of knee-joint angles (degrees)

hip_angle a tfd column of hip-joint angles (degrees)

Details

Data is also included in the **fda** package in another format.

References

Olshen, A R, Biden, N E, Wyatt, P M, Sutherland, H D (1989). “Gait Analysis and the Bootstrap.” *The Annals of Statistics*, **17**(4), 1419–1440.

Examples

```
head(gait)
```

```
growth
```

```
Berkeley growth study data
```

Description

Heights of 39 boys and 54 girls measured from age 1 to 18 years as part of the Berkeley Growth Study. The data tracks physical development over time with measurements at 31 different ages that are not equally spaced.

Usage

```
growth
```

Format

A data frame with 93 rows and 2 variables:

gender sex of the subject, a factor with levels female/male

height a tfd column of height-over-age curves (cm)

Details

Data is also included in the **fda** package in another format.

References

Ramsay, O. J, Hooker, Giles, Graves, Spencer (2009). *Functional Data Analysis with R and MATLAB*, series Use R!, 1 edition. Springer New York, New York. ISBN 978-0-387-98184-0. [doi:10.1007/9780387981857](https://doi.org/10.1007/9780387981857).

Ramsay, O. J, Silverman, W. B (2005). *Functional Data Analysis*, series Springer Series in Statistics, 2nd edition. Springer, New York. ISBN 978-0-387-40080-8.

Ramsay, O. J, Silverman, W. B (2002). *Applied Functional Data Analysis*. Springer.

Tuddenham, D R (1954). “Physical growth of California boys and girls from birth to eighteen years.” *University of California Publications in Child Development*, **1**, 183–364.

Examples

```
head(growth)
```

in_range	<i>Find out if values are inside given bounds</i>
----------	---

Description

in_range and its infix-equivalent %inr% return TRUE for all values in the numeric vector f that are within the range of values in r.

Usage

```
in_range(f, r)
```

```
f %inr% r
```

Arguments

f	a numeric vector.
r	numeric vector used to specify a range, only the minimum and maximum of r are used.

Value

a logical vector of the same length as f.

See Also

Other tidyfun utility functions: [ensure_list\(\)](#), [tf_arg\(\)](#), [tf_zoom\(\)](#), [unique_id\(\)](#)

Examples

```
in_range(1:10, c(3, 7))
1:10 %inr% c(3, 7)
```

median.tf_mv	<i>Joint depth-median for vector-valued functional data</i>
--------------	---

Description

The median of a tf_mv vector is the single *observed* curve with maximal joint depth (see [tf_depth\(\)](#)): one which.max index selects the same observation across every component, so the result is never a "chimera" stitched together from different curves. Note the deliberate divergence from [median.tf\(\)](#) on ties: the univariate median *averages* tied maximal-depth curves, but averaging components would break the observed-curve guarantee, so median.tf_mv returns the first tied curve (with a message). On tied data, median(f)\$x and median(f\$x) can therefore differ.

Usage

```
## S3 method for class 'tf_mv'
median(x, na.rm = FALSE, depth = "MBD", ...)
```

Arguments

x	a <code>tf_mv</code> vector.
na.rm	if FALSE (default), any NA observation makes the result NA; if TRUE, NA observations are dropped first.
depth	the joint depth method, see tf_depth() .
...	passed to tf_depth() .

Value

a length-1 `tf_mv`: the observed curve with maximal joint depth.

See Also

Other tidyfun summary functions: [fivenum\(\)](#), [functionwise](#), [tfsummaries](#)

Examples

```
set.seed(1)
f <- tfd_mv(list(x = tf_rgp(5), y = tf_rgp(5)))
# the joint median is the observed curve with maximal joint depth:
median(f)
tf_depth(f)
```

pinch	<i>Pinch force data</i>
-------	-------------------------

Description

Measurements of pinch force during 20 replications, with 151 observations recorded every 2 milliseconds over 300 milliseconds. The data captures the dynamics of finger pinch force applied during controlled motor tasks.

Usage

```
pinch
```

Format

A `tfd` vector of length 20 containing the pinch-force curves.

Details

Data is also included in the **fda** package in another format.

References

Ramsay, O. J, Hooker, Giles, Graves, Spencer (2009). *Functional Data Analysis with R and MATLAB*, series Use R!, 1 edition. Springer New York, New York. ISBN 978-0-387-98184-0. [doi:10.1007/9780387981857](https://doi.org/10.1007/9780387981857).

Ramsay, O. J, Silverman, W. B (2005). *Functional Data Analysis*, series Springer Series in Statistics, 2nd edition. Springer, New York. ISBN 978-0-387-40080-8.

Ramsay, O. J, Silverman, W. B (2002). *Applied Functional Data Analysis*. Springer.

Examples

```
pinch
```

```
plot.tf
```

```
base plots for tfs
```

Description

Some base functions for displaying functional data in spaghetti- (i.e., line plots) and lasagna- (i.e., heat map) flavors.

Usage

```
## S3 method for class 'tf'
plot(
  x,
  y,
  n_grid = 50,
  points = is_irreg(x),
  type = c("spaghetti", "lasagna"),
  alpha = min(1, max(0.05, 2/length(x))),
  ...
)

## S3 method for class 'tf'
lines(x, arg, n_grid = 50, alpha = min(1, max(0.05, 2/length(x))), ...)

## S3 method for class 'tf'
points(
  x,
  arg,
  n_grid = NA,
  alpha = min(1, max(0.05, 2/length(x))),
  interpolate = FALSE,
  ...
)
```

Arguments

x	a tf object.
y	(optional) numeric vector of evaluation points to use as arg for plotting (passed as the function's input grid, not the y-axis values).
n_grid	minimal size of equidistant grid used for plotting, defaults to 50. See details.
points	should the original evaluation points be marked by points? Defaults to TRUE for irregular tfd and FALSE for all others.
type	"spaghetti": line plots, "lasagna": heat maps.
alpha	alpha-value (see <code>grDevices::rgb()</code>) for noodle transparency. Defaults to 2/(no. of observations). Lower is more transparent.
...	additional arguments for <code>graphics::matplot()</code> ("spaghetti") or <code>image()</code> ("lasagna").
arg	evaluation grid (vector).
interpolate	should functions be evaluated (i.e., inter-/extrapolated) for arg for which no original data is available? Only relevant for tfd, defaults to FALSE.

Details

If no second argument `y` is given, evaluation points (`arg`) for the functions are given by the union of the `tf`'s `arg` and an equidistant grid over its domain with `n_grid` points. If you want to only see the original data for `tfd`-objects without inter-/extrapolation, use `n_grid < 1` or `n_grid = NA`.

Value

the plotted `tf`-object, invisibly.

References

Swihart, J B, Caffo, Brian, James, D B, Strand, Matthew, Schwartz, S B, Punjabi, M N (2010). "Lasagna plots: a saucy alternative to spaghetti plots." *Epidemiology (Cambridge, Mass.)*, **21**(5), 621–625.

Examples

```
f <- tfd(sin(seq(0, 2 * pi, length.out = 51)), arg = seq(0, 1, length.out = 51))
plot(f)
plot(c(f, 2 * f), type = "lasagna")
```

plot.tf_mv

*Plot vector-valued functional data***Description**

Two simple display modes for `tf_mv` objects: "facet" draws one panel per output dimension (delegating to the univariate `plot.tf()`); "trajectory" (only for $d == 2$) draws the curves in the plane, i.e. $y(t)$ against $x(t)$ – the natural view for movement data.

Usage

```
## S3 method for class 'tf_mv'
plot(x, y, ..., type = NULL)

## S3 method for class 'tf_mv'
lines(x, ..., type = NULL)

## S3 method for class 'tf_mv'
points(x, ..., type = NULL)
```

Arguments

<code>x</code>	a <code>tf_mv</code> object.
<code>y</code>	ignored.
<code>...</code>	passed to the underlying plotting calls. Per-curve graphical parameters (<code>col</code> , <code>lty</code> , <code>lwd</code> , ...) are recycled across curves.
<code>type</code>	"trajectory" or "facet". Defaults to "trajectory" for two-component ($d == 2$) objects and to "facet" otherwise.

Details

In "trajectory" mode the two components must be paired at common argument values to form $(x(t), y(t))$ points. When the components are sampled on different (or per-curve irregular) grids they are therefore evaluated on the union of their argument grids with `interpolate = TRUE` (values outside a component's observed range become NA and are skipped). For components that already share a grid this is a no-op.

Value

`x`, invisibly.

See Also

Other `tf_mv`-class: `tf_arclength()`, `tf_geom`, `tf_mv_methods`, `tfb_mfpc()`, `tfb_mv()`, `tfd_mv()`

Examples

```
arg <- seq(0, 1, length.out = 31)
xf <- tfd(t(sapply(1:5, function(i) sin(2 * pi * arg + i / 5))), arg = arg)
yf <- tfd(t(sapply(1:5, function(i) cos(2 * pi * arg + i / 5))), arg = arg)
mv <- tfd_mv(list(x = xf, y = yf))
plot(mv, type = "trajectory")
plot(mv, type = "facet")
```

```
prep_plotting_arg      Preprocess evaluation grid for plotting
```

Description

(internal function exported for re-use in upstream packages)

Usage

```
prep_plotting_arg(f, n_grid)
```

Arguments

`f` a tf-object.
`n_grid` length of evaluation grid.

Value

a semi-regular grid rounded down to appropriate resolution.

Examples

```
f <- tfd(sin(seq(0, 2 * pi, length.out = 21)), arg = seq(0, 1, length.out = 21))
prep_plotting_arg(f, n_grid = 50)
```

```
print.tf      Pretty printing and formatting for functional data
```

Description

Prints and formats tf-objects for display. See details / examples for options that give finer control.

Usage

```
## S3 method for class 'tf'
print(x, n = 6, ...)

## S3 method for class 'tfd_reg'
print(x, n = 6, ...)

## S3 method for class 'tfd_irreg'
print(x, n = 6, ...)

## S3 method for class 'tfb'
print(x, n = 5, ...)

## S3 method for class 'tf'
format(
  x,
  digits = 2,
  nsmall = 0,
  width = options()$width,
  sparkline = TRUE,
  prefix = FALSE,
  ...
)
```

Arguments

x	any R object (conceptually); typically numeric.
n	how many elements of x to print out at most, defaults to 6.
...	handed over to format.tf() .
digits	a positive integer indicating how many significant digits are to be used for numeric and complex x. The default, NULL, uses getOption("digits") . This is a suggestion: enough decimal places will be used so that the smallest (in magnitude) number has this many significant digits, and also to satisfy nsmall. (For more, notably the interpretation for complex numbers see signif .)
nsmall	the minimum number of digits to the right of the decimal point in formatting real/complex numbers in non-scientific formats. Allowed values are 0 <= nsmall <= 20.
width	default method: the <i>minimum</i> field width or NULL or 0 for no restriction. AsIs method: the <i>maximum</i> field width for non-character objects. NULL corresponds to the default 12.
sparkline	use a sparkline representation? defaults to TRUE (not available for irregular data).
prefix	prefix with names / index positions? defaults to FALSE.

Details

By default, tf objects on regular grids are shown as "sparklines" ([cli::spark_bar\(\)](#)), set sparkline = FALSE for a text representation.

Sparklines are based on running mean values of the function values, but these don't check for non-equidistant grids, so the visual impression will be misleading for very unequal grid distances.

Sparklines use `options()$width/3` bins for printing/formatting by default, use `bins` argument to set the number of bins explicitly. For `pillar::glimpse()`, we use 8 bins by default for compact display.

Value

`print`: prints out `x` and returns it invisibly.

a character representation of `x`.

Examples

```
t <- seq(0, 1, l = 201)
cosine <- lapply(1:4, \(i) cos(i * pi * t)) |> tfd(arg = t)
cosine
tf_sparsify(cosine, dropout = .8)

format(cosine, sparkline = FALSE)
format(cosine, bins = 5)
format(cosine, bins = 40)

#! very non-equidistant grids --> sparklines can mislead about actual shapes:
tfd(cosine, arg = t^3)
```

tfb

Constructors for functional data in basis representation

Description

Various constructors for `tfb`-vectors from different kinds of inputs.

Usage

```
tfb(data = data_frame0(), basis = c("spline", "fpc", "wavelet"), ...)
```

```
tfb_wavelet(data, ...)
```

```
as.tfb(data, basis = c("spline", "fpc"), ...)
```

Arguments

<code>data</code>	a matrix, data.frame or list of suitable shape, or another <code>tf</code> -object containing functional data.
<code>basis</code>	either "spline" (see <code>tfb_spline()</code> , the default) or "fpc" (see <code>tfb_fpc()</code>). (wavelet not implemented yet)
<code>...</code>	further arguments for <code>tfb_spline()</code> or <code>tfb_fpc()</code> .

Details

tfb is a wrapper for functions that set up spline-, principal component- or wavelet-based representations of functional data. For all three, the input data $x_i(t)$ are represented as weighted sums of a set of common basis functions $B_k(t); k = 1, \dots, K$ identical for all observations and weight or coefficient vectors $b_i = (b_{i1}, \dots, b_{iK})$ estimated for each observation: $x_i(t) \approx \sum_k B_k(t)b_{ik}$. Depending on the value of `basis`, the basis functions $B(t)$ will either be spline functions or the first few estimated eigenfunctions of the covariance operator of the $x(t)$ (`fpc`) or wavelets (`wavelet`).

See `tfb_spline()` for more details on spline basis representation (the default). See `tfb_fpc()` for using an functional principal component representation with an orthonormal basis estimated from the data instead.

Value

a tfb-object (or a `data.frame/matrix` for the conversion functions, obviously).

See Also

Other tfb-class: `fpc_wsvd()`, `tfb_fpc()`, `tfb_spline()`

Examples

```
arg <- seq(0, 1, length.out = 21)
x <- tfd(rbind(sin(2 * pi * arg), cos(2 * pi * arg)), arg = arg)
xb <- tfb(x, k = 8, penalized = FALSE)
xb

as.tfb(x, basis = "spline", k = 8)
```

tfb_fpc

Functional data in FPC-basis representation

Description

These functions perform a (functional) principal component analysis (FPCA) of the input data and return an `tfb_fpc` tf-object that uses the empirical eigenfunctions as basis functions for representing the data. The default (`"method = fpc_wsvd"`) uses a (truncated) weighted SVD for complete data on a common grid and a nuclear-norm regularized (truncated) weighted SVD for partially missing data on a common grid, see `fpc_wsvd()`. The latter is likely to break down for high PVE and/or high amounts of missingness.

Usage

```
tfb_fpc(data, ...)

## S3 method for class 'data.frame'
tfb_fpc(
```

```

    data,
    id = 1,
    arg = 2,
    value = 3,
    domain = NULL,
    method = fpc_wsvd,
    ...
)

## S3 method for class 'matrix'
tfb_fpc(data, arg = NULL, domain = NULL, method = fpc_wsvd, ...)

## S3 method for class 'numeric'
tfb_fpc(data, arg = NULL, domain = NULL, method = fpc_wsvd, ...)

## S3 method for class 'tf'
tfb_fpc(data, arg = NULL, method = fpc_wsvd, ...)

## Default S3 method:
tfb_fpc(data, arg = NULL, domain = NULL, method = fpc_wsvd, ...)

```

Arguments

<code>data</code>	a <code>matrix</code> , <code>data.frame</code> or list of suitable shape, or another <code>tf</code> -object containing functional data.
<code>...</code>	arguments to the method which computes the (regularized/smoothed) FPCA - see e.g. <code>fpc_wsvd()</code> . Unless set by the user, uses proportion of variance explained <code>pve = 0.995</code> to determine the truncation levels.
<code>id</code>	The name or number of the column defining which data belong to which function.
<code>arg</code>	For the list- and <code>matrix</code> -methods: numeric, or list of numerics. The evaluation grid. For the <code>data.frame</code> -method: the name/number of the column defining the evaluation grid. The <code>matrix</code> method will try to guess suitable <code>arg</code> -values from the column names of <code>data</code> if <code>arg</code> is not supplied. Other methods fall back on integer sequences (<code>1:length of data</code>) as the default if not provided.
<code>value</code>	The name or number of the column containing the function evaluations.
<code>domain</code>	range of the <code>arg</code> .
<code>method</code>	the function to use that computes eigenfunctions and scores. Defaults to <code>fpc_wsvd()</code> , which is quick and easy but returns completely unsmoothed eigenfunctions unlikely to be suited for noisy data. See details.

Details

For the FPC basis, any factorization method that accepts a `data.frame` with columns `id`, `arg`, `value` containing the functional data and returns a list with eigenfunctions and FPC scores structured like the return object of `fpc_wsvd()` can be used for the `method` argument, see example below. Note that the mean function, with a fixed "score" of 1 for all functions, is used as the first basis function for all FPC bases.

Value

an object of class `tfb_fpc`, inheriting from `tfb`. The basis used by `tfb_fpc` is a `tfd`-vector containing the estimated mean and eigenfunctions.

Methods (by class)

- `tfb_fpc(default)`: convert `tfb`: default method, returning prototype when data is `NULL`

See Also

`fpc_wsvd()` for FPCA options.

Other `tfb`-class: `fpc_wsvd()`, `tfb()`, `tfb_spline()`

Other `tfb_fpc`-class: `fpc_wsvd()`, `tfb_mfpc()`

Examples

```
set.seed(13121)
x <- tf_rgp(25, nugget = .02)
x_pc <- tfb_fpc(x, pve = .9)
x_pc
plot(x, lwd = 3)
lines(x_pc, col = 2, lty = 2)
x_pc_full <- tfb_fpc(x, pve = .995)
x_pc_full
lines(x_pc_full, col = 3, lty = 2)
# partially missing data on common grid:
x_mis <- x |> tf_sparsify(dropout = .05)
x_pc_mis <- tfb_fpc(x_mis, pve = .9)
x_pc_mis
plot(x_mis, lwd = 3)
lines(x_pc_mis, col = 4, lty = 2)
# extract FPC basis --
# first "eigenvector" in black is (always) the mean function
x_pc |> tf_basis(as_tfd = TRUE) |> plot(col = 1:5)

# Apply FPCA for sparse, irregular data using refund::fpc.sc:
set.seed(99290)
# create small, sparse, irregular data:
x_irreg <- x[1:8] |>
  tf_jiggle() |> tf_sparsify(dropout = 0.3)
plot(x_irreg)
x_df <- x_irreg |>
  as.data.frame(unnest = TRUE)
# wrap refund::fpc_sc for use as FPCA method in tfb_fpc --
# 1. define scoring function (simple weighted LS fit)
fpc_scores <- function(data_matrix, efunctions, mean, weights) {
  w_mat <- matrix(weights, ncol = length(weights), nrow = nrow(data_matrix),
    byrow = TRUE)
  w_mat[is.na(data_matrix)] <- 0
  data_matrix[is.na(data_matrix)] <- 0
  data_wc <- t((t(data_matrix) - mean) * sqrt(t(w_mat)))
```

```

    t(qr.coef(qr(efunctions), t(data_wc) / sqrt(weights)))
  }
# 2. define wrapper for fpca_sc:
fpca_sc_wrapper <- function(data, arg, pve = 0.995, ...) {
  data_mat <- tfd(data) |> as.matrix(interpolate = TRUE)
  fpca <- refund::fpca.sc(
    Y = data_mat, argvals = attr(data_mat, "arg"), pve = pve, ...
  )
  c(fpca[c("mu", "efunctions", "scores", "npc")],
    scoring_function = fpca_scores)
}
x_pc <- tfb_fpc(x_df, method = fpca_sc_wrapper)
lines(x_pc, col = 2, lty = 2)

```

tfb_mfpc

Multivariate functional principal component analysis (f: R -> R^d)

Description

tfb_mfpc() computes a *multivariate* functional principal component analysis (MFPCA) of vector-valued functional data in the sense of Happ & Greven (2018): a single set of scalar scores per curve, shared across all d components, together with vector-valued eigenfunctions $\Psi_m : \mathcal{T} \rightarrow \mathbb{R}^d$, so that $f_i(t) \approx \mu(t) + \sum_m s_{im} \Psi_m(t)$.

Usage

```

tfb_mfpc(data, ...)

## S3 method for class 'tf_mv'
tfb_mfpc(
  data,
  weights = c("inverse_variance", "snr", "equal"),
  pve = 0.995,
  npc = NULL,
  uni_pve = 0.995,
  method = fpc_wsvd,
  ...
)

## S3 method for class 'list'
tfb_mfpc(data, arg = NULL, domain = NULL, ...)

## Default S3 method:
tfb_mfpc(data, arg = NULL, domain = NULL, ...)

is_tfb_mfpc(x)

```

tf_mfpc_scores(x)

tf_mfpc_efunctions(x)

Arguments

data	a tf_d_mv() / tfb_mv object, a (named) list of univariate tf vectors, or anything tf_d_mv() accepts.
...	further arguments forwarded to the univariate method. As in tfb_mv() , a ... argument given as a list named by the component names is distributed per component.
weights	component weighting scheme for the joint analysis. Either a string – "inverse_variance" (default; $w_j = 1/\sum_l \lambda_l^{(j)}$, so each component contributes equal total variance), "snr" (signal-to-noise: retained variance over the discarded-variance tail of the univariate fit), or "equal" ($w_j = 1$) – or a numeric vector of d non-negative weights. Weights are rescaled to sum to d (so "equal" gives all-ones).
pve	proportion of variance explained used to truncate the <i>multivariate</i> components (default 0.995). Ignored if npc is given.
npc	number of multivariate FPCs to retain (overrides pve).
uni_pve	proportion of variance explained for the <i>univariate</i> FPCA of each component (default 0.995); forwarded as pve to the univariate method.
method	univariate FPCA method, see tfb_fpc() . Defaults to fpc_wsvd() .
arg	evaluation grid for raw (list/matrix/array) inputs, forwarded to tf_d_mv() .
domain	range of arg, forwarded to tf_d_mv() .
x	a tfb_mv object, ideally one returned by tfb_mfpc() .

Details

This is qualitatively different from [tfb_mv\(data, basis = "fpc"\)](#), which fits an *independent* FPCA per component (separate eigenfunctions **and** separate scores) and so cannot capture joint variation across dimensions.

The estimator first runs the univariate FPCA (see [tfb_fpc\(\)](#) / [fpc_wsvd\(\)](#)) on each component to obtain univariate scores $\xi^{(j)}$ and eigenfunctions $\phi^{(j)}$, then eigendecomposes the joint covariance of the (weighted) stacked scores. With component weights $w_j > 0$ the shared scores and multivariate eigenfunctions are

$$s_{im} = \sum_j \sqrt{w_j} \sum_l [c_m]_l^{(j)} \xi_{il}^{(j)}, \quad \Psi_m^{(j)} = \frac{1}{\sqrt{w_j}} \sum_l [c_m]_l^{(j)} \phi_l^{(j)},$$

where c_m are the eigenvectors of the weighted joint score covariance.

The returned object is a [tfb_mv\(\)](#) whose components are [tfb_fpc\(\)](#) objects sharing identical per-curve scores; cast it back to evaluations with [as.tf_d_mv\(\)](#) / [vec_cast\(\)](#), and project **new** [tf_d_mv](#) data onto the fitted basis with [tf_rebase\(\)](#). Like the univariate FPCA, the estimator targets data observed on a common grid per component; re-scoring new data evaluates it on each component's estimation grid, so new curves must be observable there.

Value

a `tfb_mv` object whose `d` components are `tfb_fpc()` objects with shared per-curve scores; `is_tfb_mfpc()` is TRUE for it. Use `tf_mfpc_scores()` and `tf_mfpc_efunctions()` to extract the shared scores and the multivariate eigenfunctions.

`is_tfb_mfpc()`: a logical flag.

`tf_mfpc_scores()`: an $n \times M$ matrix of shared multivariate FPC scores (rows = curves, columns = components).

`tf_mfpc_efunctions()`: a `tfd_mv` of length M holding the multivariate eigenfunctions Ψ_m (one "curve" per component).

References

Happ, Clara, Greven, Sonja (2018). "Multivariate functional principal component analysis for data observed on different (dimensional) domains." *Journal of the American Statistical Association*, **113**(522), 649–659.

See Also

`tfb_mv()` for independent per-component FPCA, `tfb_fpc()` / `fpc_wsvd()` for the univariate machinery.

Other `tf_mv`-class: `plot.tf_mv()`, `tf_arclength()`, `tf_geom`, `tf_mv_methods`, `tfb_mv()`, `tfd_mv()`

Other `tfb_fpc`-class: `fpc_wsvd()`, `tfb_fpc()`

Examples

```
set.seed(1)
g <- tfd_mv(list(hip = tf_rgp(20), knee = tf_rgp(20)))
m <- tfb_mfpc(g, pve = 0.99)
m
dim(tf_mfpc_scores(m))
tf_mfpc_efunctions(m)
# reconstruct and project new data:
plot(as.tfd_mv(m), type = "facet")
g_new <- tfd_mv(list(hip = tf_rgp(3), knee = tf_rgp(3)))
tf_rebase(g_new, m)
```

`tfb_mv`

Vector-valued functional data in basis representation ($f: \mathbb{R} \rightarrow \mathbb{R}^d$)

Description

`tfb_mv` is the basis-representation analogue of `tfd_mv()`: it bundles d univariate `tfb()` vectors (one per output dimension / component) into a single `vctrs` vector of vector-valued functions $f: \mathbb{R} \rightarrow \mathbb{R}^d$. Each component is fitted independently with the usual univariate `tfb()` machinery (spline or FPC basis), so all of its arguments (`k`, `bs`, `penalized`, `basis`, ...) apply per component.

Usage

```

tfb_mv(data, ...)

## S3 method for class 'tf_mv'
tfb_mv(data, basis = c("spline", "fpc"), ...)

## S3 method for class 'list'
tfb_mv(data, basis = c("spline", "fpc"), arg = NULL, domain = NULL, ...)

## Default S3 method:
tfb_mv(data, basis = c("spline", "fpc"), arg = NULL, domain = NULL, ...)

as.tfb_mv(data, ...)

## Default S3 method:
as.tfb_mv(data, ...)

```

Arguments

data	a tfd_mv() / tfb_mv object, a (named) list of univariate tf vectors, or anything tfd_mv() accepts (it is converted to tfd_mv first and then each component is expanded into a basis).
...	forwarded to the univariate tfb() constructor.
basis	spline (default) or fpc basis, see tfb() .
arg	evaluation grid for raw (list/matrix/array) inputs, forwarded to tfd_mv() .
domain	range of arg, forwarded to tfd_mv() .

Details

By default a single ... is shared across all components (every component gets the same k, bs, sp, etc.). To pass *different* basis arguments to different components, give the argument as a list named by component names – e.g. `tfb_mv(f, k = list(x = 5, y = 12))` fits component x with k = 5 and component y with k = 12. Any list-valued ... whose names do **not** match the component names is treated as a shared argument value. (Already-[tfb](#) components passed via `tfb_mv.list()` are kept as-is, which is the most permissive way to mix entirely different basis kinds across components.)

Value

a [tfb_mv](#) object.

See Also

Other [tf_mv](#)-class: [plot.tf_mv\(\)](#), [tf_arclength\(\)](#), [tf_geom](#), [tf_mv_methods](#), [tfb_mfpc\(\)](#), [tfd_mv\(\)](#)

Examples

```
traj <- tfd_mv(list(x = tf_rgp(5), y = tf_rgp(5)))
tb <- tfb_mv(traj, k = 7, verbose = FALSE)
tb
tf_ncomp(tb)
```

tfb_spline*Spline-based representation of functional data*

Description

Represent curves as a weighted sum of spline basis functions.

Usage

```
tfb_spline(data, ...)

## S3 method for class 'data.frame'
tfb_spline(
  data,
  id = 1,
  arg = 2,
  value = 3,
  domain = NULL,
  penalized = TRUE,
  global = FALSE,
  verbose = TRUE,
  ...
)

## S3 method for class 'matrix'
tfb_spline(
  data,
  arg = NULL,
  domain = NULL,
  penalized = TRUE,
  global = FALSE,
  verbose = TRUE,
  ...
)

## S3 method for class 'numeric'
tfb_spline(
  data,
  arg = NULL,
  domain = NULL,
  penalized = TRUE,
```

```
    global = FALSE,
    verbose = TRUE,
    ...
)

## S3 method for class 'list'
tfb_spline(
  data,
  arg = NULL,
  domain = NULL,
  penalized = TRUE,
  global = FALSE,
  verbose = TRUE,
  ...
)

## S3 method for class 'fd'
tfb_spline(
  data,
  arg = NULL,
  domain = NULL,
  penalized = FALSE,
  global = FALSE,
  verbose = TRUE,
  ...
)

## S3 method for class 'fdSmooth'
tfb_spline(
  data,
  arg = NULL,
  domain = NULL,
  penalized = FALSE,
  global = FALSE,
  verbose = TRUE,
  ...
)

## S3 method for class 'tfd'
tfb_spline(
  data,
  arg = NULL,
  domain = NULL,
  penalized = TRUE,
  global = FALSE,
  verbose = TRUE,
  ...
)
```

```
## S3 method for class 'tfb'
tfb_spline(
  data,
  arg = NULL,
  domain = NULL,
  penalized = TRUE,
  global = FALSE,
  verbose = TRUE,
  ...
)

## Default S3 method:
tfb_spline(
  data,
  arg = NULL,
  domain = NULL,
  penalized = TRUE,
  global = FALSE,
  verbose = TRUE,
  ...
)
```

Arguments

<code>data</code>	a matrix, data.frame or list of suitable shape, or another tf-object containing functional data.
<code>...</code>	arguments to the calls to <code>mgcv::s()</code> setting up the basis (and to <code>mgcv::magic()</code> or <code>mgcv::gam.fit()</code> if <code>penalized = TRUE</code>). Uses <code>k = 25</code> cubic regression spline basis functions (<code>bs = "cr"</code>) by default, but should be set appropriately by the user. See details and examples in the vignettes.
<code>id</code>	The name or number of the column defining which data belong to which function.
<code>arg</code>	For the list- and matrix-methods: numeric, or list of numerics. The evaluation grid. For the data.frame-method: the name/number of the column defining the evaluation grid. The matrix method will try to guess suitable arg-values from the column names of data if arg is not supplied. Other methods fall back on integer sequences (<code>1:<length of data></code>) as the default if not provided.
<code>value</code>	The name or number of the column containing the function evaluations.
<code>domain</code>	range of the arg.
<code>penalized</code>	TRUE (default) estimates regularized/penalized basis coefficients via <code>mgcv::magic()</code> or <code>mgcv::gam.fit()</code> , FALSE yields ordinary least squares / ML estimates for basis coefficients. FALSE is much faster but will overfit for noisy data if <code>k</code> is (too) large.
<code>global</code>	Defaults to FALSE. If TRUE and <code>penalized = TRUE</code> , all functions share the same smoothing parameter (see details).

verbose TRUE (default) outputs statistics about the fit achieved by the basis and other diagnostic messages.

Details

The basis to be used is set up via a call to `mgcv::s()` and all the spline bases discussed in `mgcv::smooth.terms()` are available, in principle. Depending on the value of the penalized- and global-flags, the coefficient vectors for each observation are then estimated via fitting a GAM (separately for each observation, if `!global`) via `mgcv::magic()` (least square error, the default) or `mgcv::gam()` (if a family argument was supplied) or unpenalized least squares / maximum likelihood.

After the "smoothed" representation is computed, the amount of smoothing that was performed is reported in terms of the "percentage of variability preserved", which is the variance (or the explained deviance, in the general case if family was specified) of the smoothed function values divided by the variance of the original values (the null deviance, in the general case). Reporting can be switched off with `verbose = FALSE`.

The ... arguments supplies arguments to both the spline basis (via `mgcv::s()`) and the estimation (via `mgcv::magic()` or `mgcv::gam()`), the most important arguments are:

- `k`: how many basis functions should the spline basis use, default is 25.
- `bs`: which type of spline basis should be used, the default is cubic regression splines (`bs = "cr"`)
- family argument: use this if minimizing squared errors is not a reasonable criterion for the representation accuracy (see `mgcv::family.mgcv()` for what's available) and/or if function values are restricted to be e.g. positive (`family = Gamma()/tw()/...`), in $[0, 1]$ (`family = betar()`), etc.
- `sp`: numeric value for the smoothness penalty weight, for manually setting the amount of smoothing for all curves, see `mgcv::s()`. This (drastically) reduces computation time. Defaults to -1, i.e., automatic optimization of `sp` using `mgcv::magic()` (LS fits) or `mgcv::gam()` (GLM), source code in `R/tfb-spline-utils.R`.

If `global == TRUE`, this uses a small subset of curves (10% of curves, at least 5, at most 100; non-random sample using every `j`-th curve in the data) on which smoothing parameters per curve are estimated and then takes the mean of the log smoothing parameter of those as `sp` for all curves. This is much faster than optimizing for each curve on large data sets. For very sparse or noisy curves, estimating a common smoothing parameter based on the data for all curves simultaneously is likely to yield better results, this is *not* what's implemented here.

Value

a `tfb-object`

Methods (by class)

- `tfb_spline(data.frame)`: convert data frames
- `tfb_spline(matrix)`: convert matrices
- `tfb_spline(numeric)`: convert matrices
- `tfb_spline(list)`: convert lists

- `tfb_spline(fd)`: convert `fd` objects. Almost exact re-representation for objects using Fourier- or B-spline bases, other `fda`-style bases are not implemented here.
- `tfb_spline(fdSmooth)`: convert `fdSmooth` objects. Almost exact re-representation for objects using Fourier- or B-spline bases, other `fda`-style bases are not implemented here.
- `tfb_spline(tfd)`: convert `tfd` (raw functional data)
- `tfb_spline(tfb)`: convert `tfb`: modify basis representation, smoothing.
- `tfb_spline(default)`: convert `tfb`: default method, returning prototype when data is missing

See Also

`mgcv::smooth.terms()` for spline basis options.

Other `tfb`-class: `fpc_wsvd()`, `tfb()`, `tfb_fpc()`

Examples

```
arg <- seq(0, 1, length.out = 21)
mat <- rbind(sin(2 * pi * arg), cos(2 * pi * arg))
fit <- tfb_spline(mat, arg = arg, k = 8, penalized = FALSE, verbose = FALSE)
fit
```

tfd

Constructors for vectors of "raw" functional data

Description

Various constructor methods for `tfd`-objects.

`tfd` objects contain vectors of function evaluations at observed `arg`-values, either all at the same `arg`-values (`tfd_reg`) or at different `arg`-values (`tfd_irreg`). `NA`-functions are represented by `NULL`-entries in that list.

`tfd.matrix` accepts a numeric matrix with one function per *row* (!). If `arg` is not provided, it tries to guess `arg` from the column names and falls back on `1:ncol(data)` if that fails.

`tfd.data.frame` uses the first 3 columns of data for `id` (function ID), `arg` (argument value) and `value` (function value) by default.

`tfd.list` accepts a list of vectors of identical lengths containing evaluations or a list of 2-column matrices/data.frames with `arg` in the first and evaluations in the second column

`tfd.default` returns class prototype when argument to `tfd()` is `NULL` or not a recognised class.

`as.tfd_irreg` converts regular `tfd` or `tfb` objects into irregular ones. Mainly used internally for `tf_rebase` operations etc.

Usage

```

tfd(data, ...)

## S3 method for class 'matrix'
tfd(data, arg = NULL, domain = NULL, evaluator = tf_approx_linear, ...)

## S3 method for class 'numeric'
tfd(data, arg = NULL, domain = NULL, evaluator = tf_approx_linear, ...)

## S3 method for class 'data.frame'
tfd(
  data,
  id = 1,
  arg = 2,
  value = 3,
  domain = NULL,
  evaluator = tf_approx_linear,
  ...
)

## S3 method for class 'list'
tfd(data, arg = NULL, domain = NULL, evaluator = tf_approx_linear, ...)

## S3 method for class 'tf'
tfd(data, arg = NULL, domain = NULL, evaluator = NULL, ...)

## Default S3 method:
tfd(data, arg = NULL, domain = NULL, evaluator = tf_approx_linear, ...)

as.tfd(data, ...)

as.tfd_irreg(data, ...)

```

Arguments

<code>data</code>	a <code>matrix</code> , <code>data.frame</code> or list of suitable shape, or another <code>tf</code> -object. when this argument is <code>NULL</code> (i.e. when calling <code>tfd()</code>) this returns a prototype of class <code>tfd</code> .
<code>...</code>	not used in <code>tfd</code> , except for <code>tfd.tf</code> – specify <code>arg</code> and <code>interpolate = TRUE</code> to turn an irregular <code>tfd</code> into a regular one, see examples.
<code>arg</code>	For the <code>list</code> - and <code>matrix</code> -methods: numeric, or list of numerics. The evaluation grid. For the <code>data.frame</code> -method: the name/number of the column defining the evaluation grid. The <code>matrix</code> method will try to guess suitable <code>arg</code> -values from the column names of <code>data</code> if <code>arg</code> is not supplied. Other methods fall back on integer sequences (<code>1:length of data</code>) as the default if not provided.
<code>domain</code>	range of the <code>arg</code> .
<code>evaluator</code>	a function accepting arguments <code>x</code> , <code>arg</code> , <code>evaluations</code> . See details for <code>tfd()</code> .

id	The name or number of the column defining which data belong to which function.
value	The name or number of the column containing the function evaluations.

Details

tfd-objects are list-vctrs of numeric vectors containing function evaluations.

evaluator: must be the (quoted or bare) name of a function with signature `function(x, arg, evaluations)` that returns the functions' (approximated/interpolated) values at locations `x` based on the function evaluations available at locations `arg`.

Available evaluator-functions:

- `tf_approx_linear` for linear interpolation without extrapolation (i.e., `zoo::na.approx()` with `na.rm = FALSE`) – this is the default,
 - `tf_approx_spline` for cubic spline interpolation, (i.e., `zoo::na.spline()` with `na.rm = FALSE`),
 - `tf_approx_fill_extend` for linear interpolation and constant extrapolation (i.e., `zoo::na.fill()` with `fill = "extend"`)
 - `tf_approx_locf` for "last observation carried forward" (i.e., `zoo::na.locf()` with `na.rm = FALSE`)
 - `tf_approx_nocb` for "next observation carried backward" (i.e., `zoo::na.locf()` with `na.rm = FALSE`, `fromLast = TRUE`)
- See `tf:::zoo_wrapper` and `tf::tf_approx_linear`, which is simply `zoo_wrapper(zoo::na.approx, na.rm = FALSE)`, for examples of implementations of this.

Value

a tfd-object (or a data.frame/matrix for the conversion functions, obviously).

Examples

```
# turn irregular to regular tfd by evaluating on a common grid:
```

```
f <- c(
  tf_rgp(1, arg = seq(0, 1, length.out = 11)),
  tf_rgp(1, arg = seq(0, 1, length.out = 21))
)
tfd(f, arg = seq(0, 1, length.out = 21))

set.seed(1213)
f <- tf_rgp(3, arg = seq(0, 1, length.out = 51)) |> tf_sparsify(0.9)
# does not yield regular data because linear extrapolation yields NAs
# outside observed range:
tfd(f, arg = seq(0, 1, length.out = 101))
# this "works" (but may not yield sensible values..!!) for
# e.g. constant extrapolation:
tfd(f, evaluator = tf_approx_fill_extend, arg = seq(0, 1, length.out = 101))
plot(f, col = 2)
tfd(f,
  arg = seq(0, 1, length.out = 151), evaluator = tf_approx_fill_extend
) |> lines()
```

tfd_mv

*Constructors for vector-valued functional data ($f: \mathbb{R} \rightarrow \mathbb{R}^d$)***Description**

tfd_mv represents *vector-valued* functional data – vectors of functions $f: \mathcal{T} \subset \mathbb{R} \rightarrow \mathbb{R}^d$, such as movement trajectories $(x(t), y(t))$ or other multivariate-output curves.

Usage

```
tfd_mv(data, ...)

## S3 method for class 'list'
tfd_mv(data, arg = NULL, domain = NULL, evaluator = tf_approx_linear, ...)

## S3 method for class 'array'
tfd_mv(data, arg = NULL, domain = NULL, evaluator = tf_approx_linear, ...)

## S3 method for class 'data.frame'
tfd_mv(
  data,
  id = 1,
  arg = 2,
  value = 3,
  domain = NULL,
  evaluator = tf_approx_linear,
  ...
)

## S3 method for class 'tf_mv'
tfd_mv(data, arg = NULL, domain = NULL, evaluator = NULL, ...)

## Default S3 method:
tfd_mv(data, arg = NULL, domain = NULL, ...)

as.tfd_mv(data, ...)

## Default S3 method:
as.tfd_mv(data, ...)

## S3 method for class 'tf_mv'
as.tfd_mv(data, ...)
```

Arguments

data one of: a (named) list of univariate tf vectors (used directly, one per component); a (named) list of numeric matrices / data.frames (one *per component*,

	each turned into a <code>tfd()</code>); a 3-d numeric array with dimensions <code>[curve, arg, component]</code> ; or a long data.frame with an id column, an arg column and one or more value columns (one component per value column).
<code>...</code>	forwarded to the univariate <code>tfd()</code> constructor.
<code>arg</code>	evaluation grid, see <code>tfd()</code> .
<code>domain</code>	range of arg, see <code>tfd()</code> .
<code>evaluator</code>	inter-/extrapolation function, see <code>tfd()</code> .
<code>id, value</code>	for the data.frame method: the column defining function id, the column defining the arg grid, and the (possibly several) columns containing component evaluations (one component per value column).

Details

A `tfd_mv` object of length `n` bundles `d` *univariate* `tfd()` vectors (one per output dimension / component), each of length `n`. All numeric work (evaluation, arithmetic, smoothing, ...) is delegated to these components, so regular and irregular sampling, the choice of evaluator, etc. all behave exactly as in the univariate case – and components may even live on different argument grids. Use `tfb_mv()` for a basis representation.

Value

a `tfd_mv` object (a `vctrs` vector of length `n`).

Inheritance contract

`tfd_mv` classes inherit from `"tf"` *only* for the purpose of `tf_domain()`, type predicates (`is_tf()`, `is_tf_mv()`, ...) and S4 generic reuse. **Behaviour** on `tf_mv` comes *only* from explicitly registered `.tf_mv` methods: any generic without one aborts with a classed `tf_mv_method_unimplemented` condition. The earlier promise of automatic "right thing component-wise" dispatch via inheritance was incorrect – silent fall-through produced wrong-shape results or deep internal errors, so it has been replaced with fail-fast stubs. The stubbed (i.e., *not* implemented) verbs are listed in [tf_mv_unimplemented](https://github.com/tidyfun/tf/issues/255); design of real component-wise semantics is tracked at <https://github.com/tidyfun/tf/issues/255>. When you need to *distinguish* univariate-only from any-`tf` inside a helper, use `is_tf_1d()`: it returns `TRUE` for `tfd` / `tfb` and `FALSE` for `tfd_mv` / `tfb_mv`.

See Also

`tfb_mv()` for basis representation; `tf_components()`, `tf_ncomp()` and the `$` operator to access components.

Other `tf_mv`-class: `plot.tf_mv()`, `tf_arclength()`, `tf_geom`, `tf_mv_methods`, `tfb_mfpc()`, `tfb_mv()`

Examples

```
# (a) from a (named) list of univariate tfd vectors -- one per component:
traj <- tfd_mv(list(x = tf_rgp(3), y = tf_rgp(3)))
traj
tf_ncomp(traj)
```

```

traj$x

# (b) from a list of matrices (one [curve, arg] matrix per component):
t <- seq(0, 1, length.out = 50)
mx <- matrix(sin(2 * pi * outer(1:3, t)), nrow = 3)
my <- matrix(cos(2 * pi * outer(1:3, t)), nrow = 3)
tfd_mv(list(x = mx, y = my), arg = t)

# (c) from a 3-d array with dimensions [curve, arg, component]:
arr <- array(c(mx, my), dim = c(3, 50, 2),
             dimnames = list(NULL, NULL, c("x", "y")))
tfd_mv(arr, arg = t)

# (d) from a long data.frame (id, arg, one value column per component):
df <- data.frame(
  id = rep(1:3, each = 50),
  arg = rep(t, times = 3),
  x = as.vector(t(mx)),
  y = as.vector(t(my))
)
tfd_mv(df, id = "id", arg = "arg", value = c("x", "y"))

```

tfgroupgenerics

Math, Summary and Ops Methods for tf

Description

These methods and operators mostly work arg-value-wise on tf objects, see [vctrs::vec_arith\(\)](#) etc. for implementation details.

Usage

```

## S3 method for class 'tfd'
e1 == e2

## S3 method for class 'tfd'
e1 != e2

## S3 method for class 'tfb'
e1 == e2

## S3 method for class 'tfb'
e1 != e2

## S3 method for class 'tfd'
vec_arith(op, x, y, ...)

## S3 method for class 'tfb'
vec_arith(op, x, y, ...)

```

```
## S3 method for class 'tfd'
Math(x, ...)

## S3 method for class 'tfb'
Math(x, ...)

## S3 method for class 'tf'
Summary(...)

## S3 method for class 'tfd'
cummax(...)

## S3 method for class 'tfd'
cummin(...)

## S3 method for class 'tfd'
cumsum(...)

## S3 method for class 'tfd'
cumprod(...)

## S3 method for class 'tfb'
cummax(...)

## S3 method for class 'tfb'
cummin(...)

## S3 method for class 'tfb'
cumsum(...)

## S3 method for class 'tfb'
cumprod(...)
```

Arguments

e1	an tf or a numeric vector.
e2	an tf or a numeric vector.
op	An arithmetic operator as a string.
x	a tf or numeric object.
y	a tf or numeric object.
...	tf-objects (not used for Math group generic).

Details

- Operations on tfd-objects do not extrapolate functions on a common grid first, they operate on the function at argument values that both objects have in common.

- With the exception of addition and multiplication, operations on `tfb`-objects first evaluate the data on their `arg`, perform computations on these evaluations and then convert back to an `tfb`-object, so a loss of precision should be expected – especially so for small spline bases and/or very wiggly data.
- Equality checks of functional objects are even more iffy than usual for computer math and not very reliable.
- Note that `max` and `min` are not guaranteed to be maximal/minimal over the entire domain, only at the argument values used for computation.

See examples below, many more are in `tests/testthat/test-ops.R`.

Value

a `tf`- or logical vector with the computed result.

See Also

`tf_fwise()` for scalar summaries of each function in a `tf`-vector

Examples

```
set.seed(1859)
f <- tf_rgp(4)
2 * f == f + f
sum(f) == f[1] + f[2] + f[3] + f[4]
log(exp(f)) == f
plot(f, points = FALSE)
lines(range(f), col = 2, lty = 2)

f2 <- tf_rgp(5) |> exp() |> tfb(k = 25)
layout(t(1:3))
plot(f2, col = gray.colors(5))
plot(cummin(f2), col = gray.colors(5))
plot(cumsum(f2), col = gray.colors(5))

# ?tf_integrate for integrals, ?tf_fwise for scalar summaries of each function
```

Description

These will return a `tf` object containing the respective *functional* statistic. See `tf_fwise()` for scalar summaries (e.g. `tf_fmean` for means, `tf_fmax` for max. values) of each entry in a `tf`-vector.

Usage

```
## S3 method for class 'tf'
mean(x, ...)

## S3 method for class 'tf'
median(x, na.rm = FALSE, depth = "MBD", ...)

sd(x, na.rm = FALSE)

## Default S3 method:
sd(x, na.rm = FALSE)

## S3 method for class 'tf'
sd(x, na.rm = FALSE)

var(x, y = NULL, na.rm = FALSE, use)

## Default S3 method:
var(x, y = NULL, na.rm = FALSE, use)

## S3 method for class 'tf'
var(x, y = NULL, na.rm = FALSE, use)

## S3 method for class 'tf'
summary(object, ..., depth = "MBD")

## S3 method for class 'tf_mv'
summary(object, ..., depth = "MBD")
```

Arguments

x	a tf object.
...	optional additional arguments.
na.rm	logical. Should missing values be removed?
depth	depth method used for computing the median and central region. See tf_depth() for available methods, or pass a custom depth function. Defaults to "MBD".
y	NULL (default) or a vector, matrix or data frame with compatible dimensions to x. The default is equivalent to y = x (but more efficient).
use	an optional character string giving a method for computing covariances in the presence of missing values. This must be (an abbreviation of) one of the strings "everything", "all.obs", "complete.obs", "na.or.complete", or "pairwise.complete.obs".
object	a tfd object

Value

a tf object with the computed result.
summary.tf returns a tf-vector with the mean function, the functional median, the *pointwise* min

and max of x , and the *pointwise* min and max of the central half of the functions in x , as defined by the chosen depth (default "MBD", see `tf_depth()`).

See Also

`tf_fwise()`

Other tidyfun summary functions: `fivenum()`, `functionwise`, `median.tf_mv()`

Examples

```
set.seed(123)
x <- tf_rgp(1) * 1:5
mean(x)
median(x, depth = "pointwise")
sd(x)
var(x)
summary(x)
```

tf_align

Apply warping functions to align functional data

Description

`tf_align()` applies the *inverse* warping function to unregistered data to obtain aligned (registered) functions.

Usage

```
tf_align(x, warp, ...)

## S3 method for class 'tfd'
tf_align(x, warp, ..., keep_new_arg = FALSE)

## S3 method for class 'tfb'
tf_align(x, warp, ...)
```

Arguments

<code>x</code>	tf vector of functions. For <code>tf_warp()</code> , these should be registered/aligned functions and unaligned functions for <code>tf_align()</code> .
<code>warp</code>	tf vector of warping functions used for transformation. See Details.
<code>...</code>	additional arguments passed to <code>tfd()</code> .
<code>keep_new_arg</code>	keep new arg values after (un)warping or return tfd vector on arg values of the input (default FALSE is the latter)? See Details.

Value

the aligned tf vector (registered functions)

See Also

Other registration functions: [tf_estimate_warps\(\)](#), [tf_landmarks_extrema\(\)](#), [tf_register\(\)](#), [tf_register_shape\(\)](#), [tf_registration](#), [tf_warp\(\)](#)

Examples

```
# Estimate warps, then align manually:
t <- seq(0, 2 * pi, length.out = 101)
x <- tfd(t(sapply(c(-0.3, 0, 0.3), function(s) sin(t + s))), arg = t)
warps <- tf_estimate_warps(x, method = "affine", type = "shift")
aligned <- tf_align(x, warps)
plot(aligned, col = 1:3)
```

tf_approx_linear

Inter- and extrapolation functions for tfd-objects

Description

These are exported evaluator callbacks for tfd objects. They control how function values are inter-/extrapolated to previously unseen arg values and are used by [tf_evaluate\(\)](#).

In typical use, set an evaluator when constructing a tfd (`tfd(..., evaluator = tf_approx_linear)`) or replace it later via `tf_evaluator(x) <- tf_approx_none`.

These helpers are wrappers around [zoo::na.fill\(\)](#), [zoo::na.approx\(\)](#), etc. and all share the same signature (`x, arg, evaluations`), so they can also be called directly.

The list:

- `tf_approx_linear` for linear interpolation without extrapolation (i.e., [zoo::na.approx\(\)](#) with `na.rm = FALSE`) – this is the default,
- `tf_approx_spline` for cubic spline interpolation, (i.e., [zoo::na.spline\(\)](#) with `na.rm = FALSE`),
- `tf_approx_none` in order to not inter-/extrapolate ever (i.e., [zoo::na.fill\(\)](#) with `fill = NA`)
- `tf_approx_fill_extend` for linear interpolation and constant extrapolation (i.e., [zoo::na.fill\(\)](#) with `fill = "extend"`)
- `tf_approx_locf` for "last observation carried forward" (i.e., [zoo::na.locf\(\)](#) with `na.rm = FALSE`)
- `tf_approx_nocb` for "next observation carried backward" (i.e., [zoo::na.locf\(\)](#) with `na.rm = FALSE`, `fromLast = TRUE`)

For implementing your own, see source code of `tf:::zoo_wrapper`.

Usage

```
tf_approx_linear(x, arg, evaluations)

tf_approx_spline(x, arg, evaluations)

tf_approx_none(x, arg, evaluations)

tf_approx_fill_extend(x, arg, evaluations)

tf_approx_locf(x, arg, evaluations)

tf_approx_nocb(x, arg, evaluations)
```

Arguments

x	new arg values to approximate/interpolate/extrapolate the function for.
arg	the arg values of the evaluations.
evaluations	the function values at arg.

Value

a vector of values of the function defined by the given $(x_i, f(x_i)) = (\text{arg}, \text{evaluations})$ -tuples at new argument values x.

See Also

[tfd\(\)](#)

Other tidyfun inter/extrapolation functions: [tf_evaluate\(\)](#), [tf_interpolate\(\)](#)

Examples

```
x <- tfd(matrix(c(0, 1), nrow = 1), arg = c(0, 1))
tf_evaluate(x, c(0, 0.5, 1))
tf_evaluator(x) <- tf_approx_none
tf_evaluate(x, c(0, 0.5, 1))

tf_approx_linear(
  x = c(0, 0.5, 1),
  arg = c(0, 1),
  evaluations = c(0, 1)
)
```

tf_arclength	<i>Arc length of vector-valued functional data</i>
--------------	--

Description

For a vector-valued curve $f: [a, b] \rightarrow \mathbb{R}^d$, the arc length is $\int_a^b \|f'(t)\| dt$ – the length traced out by f in $\mathbb{R}^d$.

Usage

```
tf_arclength(f, ...)

## Default S3 method:
tf_arclength(f, ...)

## S3 method for class 'tf_mv'
tf_arclength(
  f,
  arg = NULL,
  lower = tf_domain(f)[1],
  upper = tf_domain(f)[2],
  definite = TRUE,
  method = c("polyline", "derive"),
  ...
)
```

Arguments

f	a <code>tf_mv</code> object.
...	forwarded to <code>tf_integrate()</code> when <code>method = "derive"</code> .
arg, lower, upper	optional evaluation/integration grid and limits.
definite	TRUE (default) returns a numeric vector of total arc lengths per curve; FALSE returns the cumulative arc length $s(t) = \int_a^t \ f'(u)\ du$ as a univariate <code>tfd</code> .
method	"polyline" (default) or "derive".

Details

Two methods are supported:

- "polyline" (default): sum of the Euclidean lengths of the line segments between consecutive sample points (in $\mathbb{R}^d$). Each curve is evaluated on the union of its components' argument grids (or a supplied `arg`) and the segment-sum is computed in closed form. For raw `tfd_mv` data this is more accurate than "derive" because it avoids the compounding error of numerical differentiation followed by quadrature.

- "derive": composes the existing verbs – per-component differentiation (`tf_derive()`), point-wise speed `tf_speed()`, then `tf_integrate()`. Best for `tfb_mv` (analytical derivatives) or when a custom `tf_integrate(...)` argument is needed.

Value

a numeric vector (definite) or a univariate `tfd` (indefinite).

See Also

Other `tf_mv`-class: `plot.tf_mv()`, `tf_geom`, `tf_mv_methods`, `tfb_mfpc()`, `tfb_mv()`, `tfd_mv()`

Examples

```
# unit circle parameterised on [0, 1] -- arc length is 2*pi
t <- seq(0, 1, length.out = 401)
circ <- tfd_mv(list(
  x = tfd(matrix(cos(2 * pi * t), nrow = 1), arg = t),
  y = tfd(matrix(sin(2 * pi * t), nrow = 1), arg = t)
))
tf_arclength(circ)
tf_arclength(circ, lower = 0, upper = 0.25) # quarter -> pi/2
tf_arclength(circ, definite = FALSE)         # cumulative s(t)
```

tf_arg

Utility functions for tf-objects

Description

A bunch of methods & utilities that do what they say: get or set the respective attributes of a `tf`-object.

`is_tf_1d()` distinguishes *univariate* `tf` vectors (length-`n` samples of $f: \mathbb{R} \rightarrow \mathbb{R}$) from any `tf` vector. Returns `TRUE` for `tfd` / `tfb` and `FALSE` for `tfd_mv` / `tfb_mv`. Useful as a dispatch / guard predicate inside helpers that assume scalar per-arg evaluations.

Usage

`tf_arg(f)`

`tf_evaluations(f)`

`tf_count(f)`

`tf_domain(f)`

`tf_domain(x) <- value`

`tf_evaluator(f)`

```
tf_evaluator(x) <- value

tf_basis(f, as_tfd = FALSE)

tf_arg(x) <- value

## S3 replacement method for class 'tfd_irreg'
tf_arg(x) <- value

## S3 replacement method for class 'tfd_reg'
tf_arg(x) <- value

## S3 replacement method for class 'tfb'
tf_arg(x) <- value

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

## S3 method for class 'tf'
rev(x)

## S3 method for class 'tf_mv'
rev(x)

## S3 method for class 'tf'
is.na(x)

## S3 method for class 'tfd_irreg'
is.na(x)

is_tf(x)

is_tf_1d(x)

is_tfd(x)

is_reg(x)

is_tfd_reg(x)

is_irreg(x)

is_tfd_irreg(x)

is_tfb(x)

is_tfb_spline(x)
```

```
is_tfb_fpc(x)
```

```
is_tf_mv(x)
```

```
is_tfd_mv(x)
```

```
is_tfb_mv(x)
```

Arguments

f	an tf object.
x	an tf object.
value	for tf_evaluator<-: (bare or quoted) name of a function that can be used to interpolate an tfd. Needs to accept vector arguments x, arg, evaluations and return evaluations of the function defined by arg, evaluations at x. for tf_arg<-: (list of) new arg-values. for tf_domain<-: sorted numeric vector with the 2 new endpoints of the domain.
as_tfd	should the basis be returned as a tfd-vector evaluated on tf_arg(f)? Defaults to FALSE, which returns the matrix of basis functions (columns) evaluated on tf_arg(f) (rows).
object	a tfb object.
...	not used.

Value

either the respective attribute or, for setters (assignment functions), the input object with modified properties.

See Also

Other tidyfun utility functions: [ensure_list\(\)](#), [in_range\(\)](#), [tf_zoom\(\)](#), [unique_id\(\)](#)

Examples

```
x <- tf_rgp(3)
tf_arg(x)
tf_evaluations(x)
tf_count(x)
tf_domain(x)
tf_evaluator(x)
tf_evaluate(x, 0.25)
tf_evaluator(x) <- tf_approx_none
tf_evaluate(x, 0.25)
c(is_tf(x), is_tfd(x), is_reg(x), is_irreg(x))

xb <- tfb(x, k = 4, penalized = FALSE, verbose = FALSE)
```

```
tf_basis(xb)
tf_basis(xb)(c(0, .1, .2))
c(is_tfb(xb), is_tfb_spline(xb), is_tfb_fpc(xb))
```

tf_depth

Functional Data Depth

Description

Data depths for functional data. All depths are scaled so that 1 means most central and 0 means most extreme. Available methods:

Usage

```
tf_depth(x, arg, depth = "MBD", na.rm = TRUE, ...)
```

```
## S3 method for class 'matrix'
```

```
tf_depth(
  x,
  arg,
  depth = c("MBD", "MHI", "FM", "FSD", "RPD"),
  na.rm = TRUE,
  ...
)
```

```
## S3 method for class 'tf'
```

```
tf_depth(x, arg, depth = "MBD", na.rm = TRUE, ...)
```

```
## S3 method for class 'tf_mv'
```

```
tf_depth(
  x,
  arg,
  depth = c("MBD", "MHI", "FM", "FSD", "RPD"),
  na.rm = TRUE,
  weights = "equal",
  ...
)
```

Arguments

x	tf (or a matrix of evaluations).
arg	grid of evaluation points.
depth	one of "MBD", "MHI", "FM", "FSD", or "RPD".
na.rm	remove missing observations? Defaults to TRUE.
...	for "RPD": u (regularization quantile, default 0.01), n_projections (M, default 5000), n_projections_beta (L, default 500).

weights (tf_mv only) component weighting scheme for the weighted componentwise aggregation. Either "equal" (default), "inverse_variance" (weights proportional to the inverse of each component's mean pointwise variance), or a numeric vector of d strictly positive weights. Weights are normalized to sum to 1.

Details

- "MBD": Modified Band-2 Depth (default). Scale of 0 (most extreme) to 1 (most central).
- "MHI": Modified Hypograph Index. **Ranks functions from lowest (0) to highest (1)** instead of most extreme to most central! For functions that never cross: $MBD = -2(MHI - 0.5)^2 + .5$.
- "FM": Fraiman-Muniz depth. Integrates pointwise univariate halfspace depths over the domain. Scale of 0 (most extreme) to 1 (most central).
- "FSD": Functional Spatial Depth. Based on spatial signs; robust to outliers. Scale of 0 (most extreme) to 1 (most central).
- "RPD": Regularized Projection Depth. Projects curves onto random directions and computes outlyingness. Especially useful for detecting shape outliers. Scale of 0 (most extreme) to 1 (most central). **Note:** results depend on the RNG state; set a seed (e.g. `set.seed(...)`) before calling for reproducibility. Accepts additional arguments via `...`: `u` (quantile level for regularization, default 0.01), `n_projections` (M, number of projection directions, default 5000), `n_projections_beta` (L, directions for estimating regularization parameter, default 500).

Value

vector of depth values

References

- Sun, Ying, Genton, G M, Nychka, W D (2012). "Exact fast computation of band depth for large functional datasets: How quickly can one million curves be ranked?" *Stat*, **1**(1), 68–74.
- López-Pintado, Sara, Romo, Juan (2009). "On the concept of depth for functional data." *Journal of the American Statistical Association*, **104**(486), 718–734.
- López-Pintado, Sara, Romo, Juan (2011). "A half-region depth for functional data." *Computational Statistics & Data Analysis*, **55**(4), 1679–1695.
- Fraiman, Ricardo, Muniz, Graciela (2001). "Trimmed means for functional data." *Test*, **10**(2), 419–440.
- Chakraborty, Anirvan, Chaudhuri, Probal (2014). "The spatial distribution in infinite dimensional spaces and related quantiles and depths." *The Annals of Statistics*, **42**(3), 1203–1231.
- Boćinec, Filip, Nagy, Stanislav, Yeon, Hyemin (2026). "Projection depth for functional data: Practical issues, computation and applications." *arXiv preprint arXiv:2602.22877*.

See Also

Other tidyfun ordering and ranking functions: `tf_minmax`, `tf_order()`

Examples

```
x <- tf_rgp(3)/3 + 1:3
tf_depth(x, depth = "MBD")
tf_depth(x, depth = "MHI")
tf_depth(x, depth = "FM")
tf_depth(x, depth = "FSD")
```

tf_derive

*Differentiating functional data: approximating derivative functions***Description**

Derivatives of tf-objects use finite differences of the evaluations for tfd and finite differences of the basis functions for tfb.

Usage

```
tf_derive(f, arg, order = 1, ...)

## S3 method for class 'matrix'
tf_derive(f, arg, order = 1, ...)

## S3 method for class 'tfd'
tf_derive(f, arg = tf_arg(f), order = 1, ...)

## S3 method for class 'tfd_irreg'
tf_derive(f, arg, order = 1, ...)

## S3 method for class 'tfb_spline'
tf_derive(f, arg = tf_arg(f), order = 1, ...)

## S3 method for class 'tfb_fpc'
tf_derive(f, arg = tf_arg(f), order = 1, ...)
```

Arguments

f	a tf-object
arg	grid to use for the finite differences.
order	order of differentiation. Maximal value for tfb_spline is 2. For tfb_spline-objects, order = -1 yields integrals (used internally).
...	not used

Details

The derivatives of `tfd` objects use second-order accurate central differences for interior points and second-order accurate one-sided differences at boundaries, following the non-uniform grid formulas from `numpy.gradient` with `edge_order=2` (Fornberg, 1988). Domain and grid of the returned object are identical to the input. Unless the `tfd` has a rather fine and regular grid, representing the data in a suitable basis representation with `tfb()` and then computing the derivatives (or integrals) of those is usually preferable.

Note that, for spline bases like "cr" or "tp" which are constrained to begin/end linearly, computing *second* derivatives will produce artefacts at the outer limits of the functions' domain due to these boundary constraints. Basis "bs" does not have this problem for sufficiently high orders (but tends to yield slightly less stable fits).

Value

a `tf` (with the same `arg` for `tfd`-inputs, possibly different basis for `tfb`-inputs, see details).

Methods (by class)

- `tf_derive(matrix)`: row-wise finite differences
- `tf_derive(tfd)`: derivatives by finite differencing of function evaluations.
- `tf_derive(tfd_irreg)`: element-wise finite differencing for irregular grids. Falls back to `tf_derive.tfd` (interpolating to a common grid) if an explicit `arg` vector is supplied.
- `tf_derive(tfb_spline)`: derivatives by finite differencing of spline basis functions.
- `tf_derive(tfb_fpc)`: derivatives by finite differencing of FPC basis functions.

References

Fornberg, Bengt (1988). "Generation of Finite Difference Formulas on Arbitrarily Spaced Grids." *Mathematics of Computation*, **51**(184), 699–706.

See Also

Other tidyfun calculus functions: `tf_integrate()`

Examples

```
arg <- seq(0, 1, length.out = 31)
x <- tfd(rbind(arg^2, sin(2 * pi * arg)), arg = arg)
dx <- tf_derive(x)
x
dx
tf_arg(dx)
```

tf_estimate_warps	<i>Estimate warping functions for registration</i>
-------------------	--

Description

tf_estimate_warps() is the low-level workhorse for functional data registration. It estimates warping functions that align a set of functions to a template, but does *not* apply them. For a one-shot interface that also aligns the data, see [tf_register\(\)](#).

Usage

```
tf_estimate_warps(
  x,
  ...,
  template = NULL,
  method = c("srvf", "srvf_mv", "cc", "affine", "landmark"),
  max_iter = 3L,
  tol = 0.01
)
```

Arguments

x	a tf vector of functions to register.
...	additional method-specific arguments passed to backend routines (for example crit for method = "cc").
template	an optional tf vector of length 1 to use as the template. If NULL, a default template is computed (method-dependent). Not used for method = "landmark".
method	the registration method to use: • "srvf": Square Root Velocity Framework (elastic registration). For details, see fdasrvf::time_warping(). Default template is the Karcher mean. • "srvf_mv": true multivariate SRVF time registration for tf_mv objects on a regular shared grid. This method estimates one shared warp from all components jointly and does not rotate or rescale curves. • "cc": continuous-criterion registration via a tf-native dense-grid optimizer with monotone spline warps. Default template is the arithmetic mean. • "affine": affine (linear) registration with warps of the form $h(t) = a \cdot t + b$. Simpler than elastic registration, appropriate when phase variability consists only of shifts and/or uniform speed-up/slow-down. Default template is the arithmetic mean. • "landmark": piecewise-linear warps that align user-specified landmark features. Requires landmarks argument.
max_iter	integer: maximum number of Procrustes-style template refinement iterations when template = NULL. The iteration cycle is: (1) estimate template as mean of (aligned) curves, (2) register all curves to current template, (3) update template as mean of newly aligned curves, (4) repeat until convergence or max_iter

reached. Ignored when template is provided (no refinement needed) or for method = "landmark" (template not used). For method = "srvf" with template = NULL, the outer Procrustes loop is skipped regardless of max_iter because `fdasrvf::time_warping()` already computes the Karcher mean internally. Default is 3L.

tol numeric: convergence tolerance for template refinement. For method = "cc", iteration stops when the relative improvement in the registration criterion becomes negligible; for the other iterative methods, iteration stops when the relative change in the template (L2 norm) falls below tol. Default is 1e-2.

Details

For method = "cc", tf uses a tf-native dense-grid optimizer with monotone spline warps. Each warp is represented as the normalized cumulative integral of $\exp(\eta(t))$, where $\eta(t)$ is a spline with nbasis coefficients. Registration is then carried out curve-by-curve by minimizing either an integrated squared-error criterion (`crit = 1`) or the first-eigenfunction variance criterion (`crit = 2`) plus an optional spline roughness penalty (`lambda`). The outer `max_iter` loop, when template = NULL, still performs the same Procrustes-style template refinement as the other methods.

Value

tfd vector of (forward) warping functions $h_i(s) = t$ with the same length as x. Apply with `tf_align()` to obtain registered functions, or use `tf_invert()` to obtain inverse warps $h_i^{-1}(t) = s$. The returned warps carry an `attr(, "template")` with the template used (NULL for landmark registration, which has no template).

Important method-specific arguments (passed via ...)

For method = "srvf":

lambda non-negative number: penalty controlling the flexibility of warpings (default is 0 for unrestricted warps).

penalty_method cost function used to penalize warping functions. Defaults to "roughness" (norm of their second derivative), "geodesic" uses the geodesic distance to the identity and "norm" uses Euclidean distance to the identity.

For method = "cc":

nbasis integer: number of B-spline basis functions for the monotone warp basis (default 6L, minimum 2).

lambda non-negative number: roughness penalty for the warp basis (default 0 for unpenalized warping).

crit registration criterion. Defaults to 2 for the first-eigenfunction variance criterion; alternative is 1 for integrated squared error.

conv non-negative convergence tolerance for the inner optimizer. Default is 1e-4.

iterlim maximum number of inner optimization iterations per curve. Default is 20L.

For method = "affine":

type character: "shift" (translation only), "scale" (scaling only), or "shift_scale" (both). Default is "shift".

shift_range numeric(2): bounds for shift parameter. Default is $c(-\text{range}/2, \text{range}/2)$ where range is the domain width. Larger bounds allow greater shifts but may result in more NA values.

scale_range numeric(2): bounds for scale parameter. Default is $c(0.5, 2)$. Must have lower > 0 .

For method = "landmark":

landmarks **(required)** numeric matrix of landmark positions with one row per function and one column per landmark. Use `tf_landmarks_extrema()` to find peaks/valleys automatically.

template_landmarks numeric vector of target landmark positions. Default is column-wise mean of landmarks.

Author(s)

Maximilian Muecke, Fabian Scheipl, Claude Opus 4.6

See Also

Other registration functions: `tf_align()`, `tf_landmarks_extrema()`, `tf_register()`, `tf_register_shape()`, `tf_registration`, `tf_warp()`

Examples

```
# see tf_register() for full registration examples
set.seed(1)
f <- tf_rgp(5)
warps <- tf_estimate_warps(f, method = "srvf")
plot(warps)
```

tf_evaluate

Evaluate tf-vectors for given argument values

Description

Also used internally by the `[]`-operator for tf data (see `?tfbrackets`) to evaluate object, see examples.

Usage

```
tf_evaluate(object, arg, ...)

## Default S3 method:
tf_evaluate(object, arg, ...)

## S3 method for class 'tfd'
tf_evaluate(object, arg, evaluator = tf_evaluator(object), ...)

## S3 method for class 'tfb'
tf_evaluate(object, arg, ...)
```

Arguments

object	a tf, or a data.frame-like object with tf columns.
arg	optional evaluation grid (vector or list of vectors). Defaults to tf_arg(object), implicitly.
...	not used.
evaluator	optional. The function to use for inter/extrapolating the tfd. Defaults to tf_evaluator(object). See e.g. tf_approx_linear() for details.

Value

A list of numeric vectors containing the function evaluations on arg.

See Also

Other tidyfun inter/extrapolation functions: [tf_approx_linear\(\)](#), [tf_interpolate\(\)](#)

Examples

```
f <- tf_rgp(3, arg = seq(0, 1, length.out = 11))
tf_evaluate(f) |> str()
tf_evaluate(f, arg = 0.5) |> str()
# equivalent, as matrix:
f[, 0.5]
new_grid <- seq(0, 1, length.out = 6)
tf_evaluate(f, arg = new_grid) |> str()
# equivalent, as matrix:
f[, new_grid]
```

Description

Small geometric helpers defined by component-wise composition of the existing univariate Ops / Math machinery:

- `tf_norm(f)` – pointwise Euclidean norm $\|f(t)\|$;
- `tf_speed(f)` – pointwise speed $\|f'(t)\|$;
- `tf_inner(f, g)` – pointwise inner product $\langle f(t), g(t) \rangle$;
- `tf_distance(f, g)` – pointwise Euclidean distance $\|f(t) - g(t)\|$;
- `tf_tangent(f)` – unit tangent $f'(t)/\|f'(t)\|$ (undefined where the speed is zero – callers get NaNs there);
- `tf_reparam_arclength(f)` – re-parametrize the curve at constant speed (i.e. by its normalized cumulative arc length).

Usage

```
tf_norm(f)
```

```
## Default S3 method:
```

```
tf_norm(f)
```

```
## S3 method for class 'tf'
```

```
tf_norm(f)
```

```
## S3 method for class 'tf_mv'
```

```
tf_norm(f)
```

```
tf_speed(f)
```

```
tf_inner(f, g)
```

```
## Default S3 method:
```

```
tf_inner(f, g)
```

```
## S3 method for class 'tf'
```

```
tf_inner(f, g)
```

```
## S3 method for class 'tf_mv'
```

```
tf_inner(f, g)
```

```
tf_distance(f, g)
```

```
tf_tangent(f)
```

```
## Default S3 method:
tf_tangent(f)

## S3 method for class 'tf'
tf_tangent(f)

## S3 method for class 'tf_mv'
tf_tangent(f)

tf_reparam_arclength(f)
```

Arguments

`f, g` `tf_mv` objects, or univariate `tf` (`tfd`/`tfb`) objects (with identical `d` and component names where two `tf_mv` arguments are required).

Details

These also apply to *univariate* `tfd`/`tfb` (treated as scalar-valued curves $f : T \rightarrow \mathbb{R}$), where they reduce to their one-dimensional specializations: $\|f(t)\| = |f(t)|$, $\langle f(t), g(t) \rangle = f(t) g(t)$, and the unit tangent $f'(t)/|f'(t)| = \text{sign}(f'(t))$.

Value

a univariate `tfd` for `tf_norm`/`tf_speed`/`tf_inner`/`tf_distance`; `tf_tangent` returns a `tf_mv` (or a univariate `tf` for univariate input) and `tf_reparam_arclength` a `tf_mv`.

See Also

Other `tf_mv`-class: [plot.tf_mv\(\)](#), [tf_arclength\(\)](#), [tf_mv_methods](#), [tfb_mfpc\(\)](#), [tfb_mv\(\)](#), [tfd_mv\(\)](#)

Examples

```
set.seed(1)
f <- tfd_mv(list(x = tf_rgp(2), y = tf_rgp(2)))
tf_norm(f)
tf_speed(f)
tf_distance(f, tfd_mv(list(x = tf_rgp(2), y = tf_rgp(2))))
# univariate: tf_norm reduces to the pointwise absolute value
u <- tf_rgp(2)
tf_norm(u)
tf_inner(u, tf_rgp(2))
```

tf_integrate	<i>Integrals and anti-derivatives of functional data</i>
--------------	--

Description

Integrals of tf-objects are computed by simple quadrature (trapezoid rule). By default the scalar definite integral $\int_{lower}^{upper} f(s)ds$ is returned (option `definite = TRUE`), alternatively for `definite = FALSE` the *anti-derivative* on $[lower, upper]$, e.g. a tfd or tfb object representing $F(t) \approx \int_{lower}^t f(s)ds$, for $t \in [lower, upper]$, is returned.

Usage

```
tf_integrate(f, arg, lower, upper, ...)
```

```
## Default S3 method:
```

```
tf_integrate(f, arg, lower, upper, ...)
```

```
## S3 method for class 'tfd'
```

```
tf_integrate(
  f,
  arg = tf_arg(f),
  lower = tf_domain(f)[1],
  upper = tf_domain(f)[2],
  definite = TRUE,
  ...
)
```

```
## S3 method for class 'tfb'
```

```
tf_integrate(
  f,
  arg = tf_arg(f),
  lower = tf_domain(f)[1],
  upper = tf_domain(f)[2],
  definite = TRUE,
  ...
)
```

Arguments

f	a tf-object
arg	(optional) grid to use for the quadrature.
lower	lower limits of the integration range. For <code>definite = TRUE</code> , this can be a vector of the same length as f.
upper	upper limits of the integration range (but see <code>definite</code> arg / description). For <code>definite = TRUE</code> , this can be a vector of the same length as f.
...	not used

`definite` should the definite integral be returned (default) or the antiderivative. See description.

Details

When `f` is irregular **and** `lower` / `upper` are not supplied explicitly, they default to each curve's own observed arg range (i.e., the range of its `tf_arg()` values) rather than the (shared) domain endpoints; for regular `tfd` the defaults remain the domain endpoints. Without this per-curve default, curves that do not span the full domain would silently NA-poison the trapezoidal sum, because the default linear evaluator does not extrapolate. Pass explicit `lower` / `upper` to integrate over a fixed sub-interval, or switch to an extrapolating evaluator (e.g. `tf_approx_fill_extend()`) to integrate over the full domain.

Value

For `definite = TRUE`, the definite integrals of the functions in `f`. For `definite = FALSE` and `tf`-inputs, a `tf` object containing their anti-derivatives

See Also

Other tidyfun calculus functions: `tf_derive()`

Examples

```
arg <- seq(0, 1, length.out = 11)
x <- tfd(rbind(arg, arg^2), arg = arg)
tf_integrate(x)
anti <- tf_integrate(x, definite = FALSE)
tf_arg(anti)
```

<code>tf_interpolate</code>	<i>Re-evaluate tf-objects on a new grid of argument values.</i>
-----------------------------	---

Description

Change the internal representation of a `tf`-object so that it uses a different grid of argument values (`arg`). Useful for

- thinning out dense grids to make data smaller
- filling out sparse grids to make derivatives/integrals and locating extrema or zero crossings more accurate (... *if* the interpolation works well ...)
- making irregular functional data into (more) regular data.

For `tfd`-objects, this is just syntactic sugar for `tfd(object, arg = arg)`. To inter/extrapolate more reliably and avoid NAs, call `tf_interpolate` with `evaluator = tf_approx_fill_extend`.

For `tfb`-objects, this re-evaluates basis functions on the new grid which can speed up subsequent computations if they all use that grid. NB: **To reliably impute very irregular data on a regular, common grid, you'll be better off doing FPCA-based imputation or other model-based approaches in most cases.**

Usage

```
tf_interpolate(object, arg, ...)

## S3 method for class 'tfb'
tf_interpolate(object, arg, ...)

## S3 method for class 'tfd'
tf_interpolate(object, arg, ...)

## S3 method for class 'tf_mv'
tf_interpolate(object, arg, ...)
```

Arguments

object	an object inheriting from tf.
arg	a vector of argument values on which to evaluate the functions in object.
...	additional arguments handed over to tfd or tfb, for the construction of the returned object.

Value

a tfd or tfb object on the new grid given by arg.

See Also

[tf_rebase\(\)](#), which is more general.

Other tidyfun inter/extrapolation functions: [tf_approx_linear\(\)](#), [tf_evaluate\(\)](#)

Examples

```
# thinning out a densely observed tfd
dense <- tf_rgp(10, arg = seq(0, 1, length.out = 1001))
less_dense <- tf_interpolate(dense, arg = seq(0, 1, length.out = 101))
dense
less_dense

# filling out sparse data (use a suitable evaluator-function!)
sparse <- tf_rgp(10, arg = seq(0, 5, length.out = 11))
plot(sparse, points = TRUE)
# change evaluator for better interpolation
tfd(sparse, evaluator = tf_approx_spline) |>
  tf_interpolate(arg = seq(0, 5, length.out = 201)) |>
  lines(col = 2, lty = 2)

set.seed(1860)
sparse_irregular <- tf_rgp(5) |>
  tf_sparsify(0.5) |>
  tf_jiggle()
tf_interpolate(sparse_irregular, arg = seq(0, 1, length.out = 51))
```

tf_invert	<i>Invert a tf vector</i>
-----------	---------------------------

Description

Computes the functional inverse of each function in the `tf` vector, such that if $y = f(x)$, then $x = f^{-1}(y)$.

Usage

```
tf_invert(x, ...)
```

Arguments

<code>x</code>	a <code>tf</code> vector.
<code>...</code>	optional arguments for the returned object, see tfd() / tfb()

Value

a `tf` vector of the inverted functions.

Examples

```
arg <- seq(0, 2, length.out = 50)
x <- tfd(rbind(2 * arg, arg^2), arg = arg)
x_inv <- tf_invert(x)
layout(t(1:2))
plot(x, main = "original functions", ylab = "")
plot(x_inv, main = "inverted functions", ylab = "", points = FALSE)
```

tf_jiggle	<i>Make a tf (more) irregular</i>
-----------	-----------------------------------

Description

Randomly create some irregular functional data from regular ones.

- **jiggle** it by randomly moving around its `arg`-values inside the intervals defined by its grid neighbors on the original argument grid.
- **sparsify** it by removing $(100 * \text{dropout})\%$ of the function values

Usage

```
tf_jiggle(f, amount = 0.4, ...)

## S3 method for class 'tf_mv'
tf_jiggle(f, amount = 0.4, same_arg = TRUE, ...)

tf_sparsify(f, dropout = 0.5, ...)

## S3 method for class 'tf_mv'
tf_sparsify(f, dropout = 0.5, same_arg = TRUE, ...)
```

Arguments

<code>f</code>	a tfd object.
<code>amount</code>	how far away from original grid points can the jiggled grid points lie, at most (relative to original distance to neighboring grid points). Defaults to at most 40% (0.4) of the original grid distances. Must be lower than 0.5.
<code>...</code>	additional args for the returned tfd in <code>tf_jiggle</code> .
<code>same_arg</code>	for <code>tf_mv</code> objects, should all components receive the same random argument-grid changes? Defaults to TRUE; use FALSE to jitter or sparsify each component independently.
<code>dropout</code>	what proportion of values of <code>f</code> to drop, on average. Defaults to half.

Value

an (irregular) tfd object.

See Also

Other tidyfun RNG functions: [tf_rgp\(\)](#)

Examples

```
set.seed(1)
(x <- tf_rgp(2, arg = 21L))
(x_jig <- tf_jiggle(x, amount = 0.2))
(x_sp <- tf_sparsify(x, dropout = 0.3))
c(is_irreg(x_jig), is_irreg(x_sp))
```

tf_landmarks_extrema *Find Extrema Locations in Functional Data*

Description

Find landmark locations for registration

Usage

```
tf_landmarks_extrema(x, which = "all", threshold = 0.5, boundary_tol = NULL)
```

Arguments

x	a tf vector.
which	character: which features to detect. Either "all" (maxima, minima, and zero crossings), "both" (maxima and minima), or any subset of c("max", "min", "zero").
threshold	numeric in (0, 1]: minimum proportion of curves that must contain a feature for it to be retained as a landmark. Defaults to 0.5.
boundary_tol	numeric: features within this distance of the domain boundary are dropped (they are redundant with the boundary anchors in landmark registration). Defaults to 2x the grid spacing. Set to 0 to keep all features.

Details

Detects local maxima, minima, and/or zero crossings in each function and returns a landmark matrix suitable for `tf_register()` with `method = "landmark"`. Uses position-based clustering across curves to establish feature correspondence and majority-count filtering to discard unstable landmarks.

Value

A numeric matrix with one row per function and one column per landmark, sorted left-to-right on the domain. Has attribute "feature_types" (character vector of "max", "min", or "zero" for each column). Contains NA where a curve is missing a landmark.

See Also

`tf_register()` with `method = "landmark"`

Other registration functions: `tf_align()`, `tf_estimate_warps()`, `tf_register()`, `tf_register_shape()`, `tf_registration`, `tf_warp()`

Examples

```
t <- seq(0, 1, length.out = 101)
x <- tfd(t(sapply(c(0.3, 0.5, 0.7), function(s) dnorm(t, s, 0.1))), arg = t)
tf_landmarks_extrema(x, "max")
tf_landmarks_extrema(x, "both")
```

tf_minmax

*Depth-based minimum, maximum and range for tf vectors***Description**

By default, min, max, and range compute **pointwise** extremes (the existing behaviour). When a depth argument is supplied, they instead return the most extreme / most central observation according to the chosen depth. For the default "MHI" depth this gives the lowest / highest function in an up-down sense.

Usage

```
## S3 method for class 'tf'
min(..., na.rm = FALSE, depth = NULL)

## S3 method for class 'tf'
max(..., na.rm = FALSE, depth = NULL)

## S3 method for class 'tf'
range(..., na.rm = FALSE, depth = NULL)
```

Arguments

...	tf objects (and na.rm for the pointwise default).
na.rm	logical; passed on to the pointwise summary or used to filter NAs before computing depth.
depth	depth method to use. NULL (default) gives the pointwise min/max/range. Supply a depth name (e.g. "MHI") or a custom depth function for depth-based selection.

Value

a tf object.

See Also

[tf_depth\(\)](#), [rank.tf\(\)](#)

Other tidyfun ordering and ranking functions: [tf_depth\(\)](#), [tf_order\(\)](#)

Examples

```
x <- tf_rgp(5) + 1:5
# pointwise (default):
min(x)
max(x)
# depth-based:
min(x, depth = "MHI")
max(x, depth = "MHI")
```

Description

Utilities for `tf_mv` objects (see [tfd_mv\(\)](#) / [tfb_mv\(\)](#)). `tf_ncomp()` returns the number of output dimensions d , `tf_components()` the list of the d underlying univariate `tf` vectors, and `tf_component()` extracts or replaces a single one (also available via the `$` operator, e.g. `f$x`).

Usage

```
tf_ncomp(f)

tf_components(f)

tf_component(f, which)

tf_component(f, which) <- value
```

Arguments

<code>f</code>	a <code>tf_mv</code> object.
<code>which</code>	a component name or index.
<code>value</code>	a univariate <code>tf</code> vector (replacement) of matching length and domain.

Details

Most univariate `tf` verbs also work on `tf_mv` objects by acting on each component: [tf_rebase\(\)](#) (and hence `tfd_mv/tfb_mv` conversion), [tf_derive\(\)](#), [tf_integrate\(\)](#) (definite integrals return an $n \times d$ matrix), [tf_smooth\(\)](#) and [tf_zoom\(\)](#). Registration ([tf_register\(\)](#) / [tf_estimate_warps\(\)](#) / [tf_warp\(\)](#) / [tf_align\(\)](#)) estimates a *single, shared* time-warp per curve and applies it jointly to every component. The registration signal is, by default, the first component; use `ref_component` to pick another component (by name/index), "norm" for the pointwise Euclidean norm, or a function mapping the `tf_mv` to a univariate `tf` vector.

`is.na()` flags a curve as missing if **any** of its components is missing (the union, not the intersection), which also drives the `na.rm` behaviour of [mean\(\)](#) / [median\(\)](#) etc.

Value

`tf_ncomp()`: an integer. `tf_components()`: a named list of `tf` vectors. `tf_component()`: a single univariate `tf` vector.

See Also

Other `tf_mv`-class: [plot.tf_mv\(\)](#), [tf_arclength\(\)](#), [tf_geom](#), [tfb_mfpc\(\)](#), [tfb_mv\(\)](#), [tfd_mv\(\)](#)

Examples

```
f <- tfd_mv(list(x = tf_rgp(3), y = tf_rgp(3)))
tf_ncomp(f)
tf_components(f)
tf_component(f, "y")
f$y
```

tf_order

Rank, order and sort tf vectors

Description

These methods use `tf_depth()` to rank, order, and sort functional data. By default they use the modified hypograph index ("MHI") which provides an up-down ordering (lowest to highest). You can also use any of the other depth methods available via `tf_depth()`, or supply a custom depth function.

Usage

```
rank(
  x,
  na.last = TRUE,
  ties.method = c("average", "first", "last", "random", "max", "min"),
  ...
)

## Default S3 method:
rank(
  x,
  na.last = TRUE,
  ties.method = c("average", "first", "last", "random", "max", "min"),
  ...
)

## S3 method for class 'tf'
rank(
  x,
  na.last = TRUE,
  ties.method = c("average", "first", "last", "random", "max", "min"),
  depth = "MHI",
  ...
)

## S3 method for class 'tf'
xtfrm(x)

## S3 method for class 'tf'
```

```

sort(x, decreasing = FALSE, na.last = NA, depth = "MHI", ...)

tf_order(f, ...)

## Default S3 method:
tf_order(f, ...)

## S3 method for class 'tf'
tf_order(f, depth = "MHI", ...)

## S3 method for class 'tf_mv'
tf_order(f, by = "norm", ...)

```

Arguments

<code>x</code>	a <code>tf</code> vector.
<code>na.last</code>	for handling of NAs; see <code>base::rank()</code> and <code>base::sort()</code> .
<code>ties.method</code>	a character string for handling ties; see <code>base::rank()</code> .
<code>...</code>	passed to <code>tf_depth()</code> (e.g. <code>arg</code>).
<code>depth</code>	the depth function to use for ranking. One of the depths available via <code>tf_depth()</code> (default: "MHI") or a function that takes a <code>tf</code> vector and returns a numeric vector of depth values.
<code>decreasing</code>	logical. Should the sort be decreasing?
<code>f</code>	a <code>tf</code> or <code>tf_mv</code> vector (for <code>tf_order</code>).
<code>by</code>	(<code>tf_mv</code> only) the scalar reduction to order by: "norm" (the default, uses <code>tf_norm(f)</code>) or the name of a component.

Details

`rank` assigns ranks based on depth values: lower depth values get lower ranks. For "MHI" this gives an ordering from lowest to highest function. For centrality-based depths ("MBD", "FM", "FSD", "RPD"), the most extreme function gets rank 1 and the most central gets the highest rank.

`tf_order` returns the permutation which rearranges `x` into ascending order according to depth. For vector-valued (`tf_mv`) data there is no canonical total order on R^d : `tf_order.tf_mv()` requires an explicit scalar reduction via `by` (either "norm" for `tf_norm(f)`, or a component name), and then applies the univariate depth order to that reduction.

`sort.tf` returns the sorted `tf` vector.

`xtfrm.tf` returns a numeric vector of MHI depth values, enabling `base::order` and `base::rank` to work on `tf` vectors.

Value

`rank`: a numeric vector of ranks.
`tf_order`: an integer vector of indices.
`sort.tf`: a sorted `tf` vector.
`xtfrm.tf`: a numeric vector of depth values.

See Also

`tf_depth()`, `min.tf()`, `max.tf()`

Other tidyfun ordering and ranking functions: `tf_depth()`, `tf_minmax`

Examples

```
x <- tf_rgp(5) + 1:5
rank(x)
order(x)
sort(x)
# use a centrality-based depth instead:
rank(x, depth = "MBD")
```

tf_rebase

Change (basis) representation of a tf-object

Description

Apply the representation of one tf-object to another; i.e. re-express it in the other's basis, on its grid, etc.

Useful for making different functional data objects compatible so they can be combined, compared or computed with.

Usage

```
tf_rebase(object, basis_from, arg = tf_arg(basis_from), ...)

## S3 method for class 'tfd'
tf_rebase(object, basis_from, arg = tf_arg(basis_from), ...)

## S3 method for class 'tfb'
tf_rebase(object, basis_from, arg = tf_arg(basis_from), ...)
```

Arguments

<code>object</code>	a tf object whose representation should be changed.
<code>basis_from</code>	the tf object with the desired basis, arg, evaluator, etc.
<code>arg</code>	optional new arg values, defaults to those of <code>basis_from</code> .
<code>...</code>	forwarded to the tfb or tfd constructors.

Details

This uses double dispatch (S3) internally, so the methods defined below are themselves generics for methods `tf_rebase.tfd.tfd`, `tf_rebase.tfd.tfb_spline`, `tf_rebase.tfd.tfb_fpc`, `tf_rebase.tfb.tfd`, `tf_rebase.tfb.tfb` that dispatch on `object_from`.

Value

a tf-vector containing the data of object in the same representation as basis_from (potentially modified by the arguments given in ...).

Methods (by class)

- `tf_rebase(tfd)`: re-express a tfd-vector in the same representation as some other tf-vector
- `tf_rebase(tfb)`: re-express a tfb-vector in the same representation as some other tf-vector.

Examples

```
x <- tf_rgp(3)
xb <- tfb(x, k = 8, penalized = FALSE, verbose = FALSE)
tf_rebase(tf_rgp(3), xb)
```

tf_register

Register / align a tf vector against a template function

Description

`tf_register()` is the high-level entry point for functional data registration. It estimates warping functions, applies them to align the data, and returns a [tf_registration](#) result object containing the aligned curves, inverse warping functions (observed to aligned time), and template. Use [tf_aligned\(\)](#), [tf_inv_warps\(\)](#), and [tf_template\(\)](#) to extract components.

Usage

```
tf_register(
  x,
  ...,
  template = NULL,
  method = c("srvf", "srvf_mv", "cc", "affine", "landmark"),
  max_iter = 3L,
  tol = 0.01,
  store_x = TRUE
)
```

Arguments

<code>x</code>	a tf vector of functions to register.
<code>...</code>	additional method-specific arguments passed to backend routines (for example <code>crit</code> for <code>method = "cc"</code>). See tf_estimate_warps() for method-specific argument documentation.
<code>template</code>	an optional tf vector of length 1 to use as the template. If <code>NULL</code> , a default template is computed (method-dependent). Not used for <code>method = "landmark"</code> .

method	the registration method to use: • "srvf": Square Root Velocity Framework (elastic registration). • "srvf_mv": true multivariate SRVF time registration for <code>tf_mv</code> objects on a regular shared grid. • "cc": continuous-criterion registration via a tf-native dense-grid optimizer with monotone spline warps. • "affine": affine (linear) registration. • "landmark": piecewise-linear warps aligning user-specified landmarks.
max_iter	integer: maximum Procrustes-style template refinement iterations. Default 3L.
tol	numeric: convergence tolerance for template refinement. Default 1e-2.
store_x	logical: store original data in the result object? Default TRUE. Set to FALSE to save memory.

Details

For a lower-level interface that returns only warping functions (without performing alignment), see [tf_estimate_warps\(\)](#).

Value

A [tf_registration](#) object. Access components with [tf_aligned\(\)](#), [tf_inv_warps\(\)](#), [tf_template\(\)](#).

Important method-specific arguments (passed via ...)

For method = "srvf":

`lambda` non-negative number: penalty controlling the flexibility of warpings (default is 0 for unrestricted warps).

`penalty_method` cost function used to penalize warping functions. Defaults to "roughness" (norm of their second derivative), "geodesic" uses the geodesic distance to the identity and "norm" uses Euclidean distance to the identity.

For method = "cc":

`nbasis` integer: number of B-spline basis functions for the monotone warp basis (default 6L, minimum 2).

`lambda` non-negative number: roughness penalty for the warp basis (default 0 for unpenalized warping).

`crit` registration criterion. Defaults to 2 for the first-eigenfunction variance criterion; alternative is 1 for integrated squared error.

`conv` non-negative convergence tolerance for the inner optimizer. Default is 1e-4.

`iterlim` maximum number of inner optimization iterations per curve. Default is 20L.

For method = "affine":

`type` character: "shift" (translation only), "scale" (scaling only), or "shift_scale" (both). Default is "shift".

shift_range numeric(2): bounds for shift parameter. Default is `c(-range/2, range/2)` where range is the domain width. Larger bounds allow greater shifts but may result in more NA values.

scale_range numeric(2): bounds for scale parameter. Default is `c(0.5, 2)`. Must have lower > 0.

For method = "landmark":

landmarks **(required)** numeric matrix of landmark positions with one row per function and one column per landmark. Use `tf_landmarks_extrema()` to find peaks/valleys automatically.

template_landmarks numeric vector of target landmark positions. Default is column-wise mean of landmarks.

Author(s)

Maximilian Muecke, Fabian Scheipl, Claude Opus 4.6

References

Ramsay JO, Hooker G, Graves S (2009). *Functional Data Analysis with R and MATLAB*. Springer, New York. doi:10.1007/9780387981857.

Srivastava A, Wu W, Kurtek S, Klassen E, Marron JS (2011). "Registration of Functional Data Using Fisher-Rao Metric." *arXiv:1103.3817*.

Tucker JD, Wu W, Srivastava A (2013). "Generative models for functional data using phase and amplitude separation." *Computational Statistics & Data Analysis*, **61**, 50–66. doi:10.1016/j.csda.2012.12.001.

See Also

Other registration functions: `tf_align()`, `tf_estimate_warps()`, `tf_landmarks_extrema()`, `tf_register_shape()`, `tf_registration`, `tf_warp()`

Examples

```
# Elastic registration (SRVF method)
height_female <- subset(growth, gender == "female", select = height, drop = TRUE)
growth_female <- tf_derive(height_female) |> tfd(arg = seq(1.125, 17.8), l = 101)
reg <- tf_register(growth_female)
layout(t(1:3))
plot(growth_female, xlab = "Chronological Age", ylab = "Growth Rate (cm/year)")
plot(tf_inv_warps(reg), xlab = "Chronological Age", ylab = "Biological Age")
plot(tf_aligned(reg), xlab = "Biological Age", ylab = "Growth Rate (cm/year)")

# Affine registration (shift only)
t <- seq(0, 2 * pi, length.out = 101)
x <- tfd(t(sapply(c(-0.3, 0, 0.3), function(s) sin(t + s))), arg = t)
reg <- tf_register(x, method = "affine", type = "shift")
plot(tf_aligned(reg), col = 1:3)

# Landmark registration
peaks <- tf_landmarks_extrema(x, "max")
```

```
reg <- tf_register(x, method = "landmark", landmarks = peaks)
plot(tf_aligned(reg), col = 1:3)
```

tf_register_shape	<i>Register vector-valued curves in elastic shape space</i>
-------------------	---

Description

tf_register_shape() aligns vector-valued tf_mv curves with SRVF curve alignment, optionally allowing rotations and rescaling in addition to time warping. Unlike [tf_register\(\)](#), this is a shape-registration interface: the aligned curves live in centered shape space and the result stores rotations and scale factors.

Usage

```
tf_register_shape(
  x,
  ...,
  template = NULL,
  max_iter = 3L,
  tol = 0.01,
  rotation = TRUE,
  scale = TRUE,
  mode = c("O", "C"),
  lambda = 0,
  store_x = TRUE
)
```

Arguments

x	a regular shared-grid tfd_mv object.
...	additional arguments passed to fdasrvf alignment routines, such as ncores or verbose.
template	optional length-one tf_mv template. If NULL, a template is refined from the first curve by iterative group alignment.
max_iter	integer: maximum template refinement iterations when template = NULL. Default 3L.
tol	numeric: relative template convergence tolerance.
rotation	logical: allow rotations? Default TRUE.
scale	logical: allow scale changes? Default TRUE.
mode	character: curve mode for fdasrvf. Only open curves ("O", the default) are currently supported; "C" (closed curves) is reserved for a future release (see Details).
lambda	numeric: non-negative elastic penalty passed to the fdasrvf alignment routines. Larger values penalise warping. Default 0.
store_x	logical: store original data in the result object?

Details

When `scale = TRUE` the aligned curves returned by `tf_aligned()` are renormalised to a common (mean) arc length so that congruent shapes overlay. The per-curve factors returned by `tf_scales()` are the sizes that were removed: multiplying an aligned curve by its scale factor rescales it back to the corresponding input curve's arc length, so a value > 1 means the input curve was larger than the shared aligned size and < 1 means it was smaller. With `scale = FALSE` warping and rotation preserve arc length, so all factors are 1. With `template = NULL` the returned `tf_template()` is the empirical mean of the aligned shape-space curves rather than any single input curve.

Only open curves (`mode = "O"`) are supported. Closed curves (`mode = "C"`) additionally optimise over a circular seed shift that the returned warping functions do not represent, which would make the stored warps inconsistent with the aligned curves; `mode = "C"` is therefore rejected for now.

Value

A `tf_shape_registration` object. Access aligned curves with `tf_aligned()`, inverse warps with `tf_inv_warps()`, the template with `tf_template()`, rotations with `tf_rotations()`, and scales with `tf_scales()`.

See Also

Other registration functions: `tf_align()`, `tf_estimate_warps()`, `tf_landmarks_extrema()`, `tf_register()`, `tf_registration`, `tf_warp()`

Examples

```
t <- seq(0, 1, length.out = 51)
base <- rbind(t, t^2)
beta <- array(NA_real_, dim = c(3, length(t), 2))
for (i in 1:3) {
  beta[i,, 1] <- base[1, ]
  beta[i,, 2] <- base[2, ]
}
curves <- tfd_mv(beta, arg = t)
reg <- tf_register_shape(curves, max_iter = 1)
tf_rotations(reg)
tf_scales(reg)
```

tf_registration

Registration Result Object

Description

`tf_registration` objects store the result of `tf_register()`, including the aligned (registered) curves, estimated inverse warping functions h_i^{-1} (observed $\rightarrow$ aligned time), and the template used. Use accessors `tf_aligned()`, `tf_inv_warps()`, and `tf_template()` to extract components. `tf_shape_registration` objects, returned by `tf_register_shape()`, extend this structure with shape-space rotations and scale factors. Use `tf_rotations()` and `tf_scales()` to extract those components.

Usage

```
tf_aligned(x)

tf_inv_warps(x)

tf_template(x)

tf_rotations(x)

tf_scales(x)

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

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

## S3 method for class 'tf_registration'
summary(object, ...)

## S3 method for class 'summary.tf_registration'
print(x, ...)

## S3 method for class 'tf_shape_registration'
summary(object, ...)

## S3 method for class 'summary.tf_shape_registration'
print(x, ...)

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

## S3 method for class 'tf_registration'
x[i]

## S3 method for class 'tf_registration'
length(x)

## S3 method for class 'tf_shape_registration'
x[i]
```

Arguments

x	a tf_registration or tf_shape_registration object
...	additional arguments (currently unused)
object	a tf_registration object
i	index for subsetting (integer, logical, or character)

Value

For `tf_registration` objects: a list with entries `registered` (tf-vector of aligned/registered functions from `x`), `inv_warps` (inverse warping functions aligning `x` to the template function), the template function, the original data `x` (if `store_x = TRUE` was used in `tf_register()`), and the call to `tf_register()` that created the object. `tf_shape_registration` objects additionally contain the forward warps, rotations, and scales. Accessors return the respective component.

Summary diagnostics

`summary()` computes per-curve diagnostics for assessing registration quality and prints their averages and/or deciles. The printed output contains four sections:

Amplitude variance reduction (only if `store_x = TRUE`): the proportion of pointwise variance removed by registration, computed as $1 - \bar{V}_{\text{registered}} / \bar{V}_{\text{original}}$ where $\bar{V}$ is the mean (across the domain) of the pointwise variance (across curves). Values near 1 indicate that registration removed most of the original variability; values near 0 indicate little change; negative values indicate that registration *increased* variability (a sign that something went wrong).

Warp deviation from identity (deciles across curves): each curve's inverse warping function h_i^{-1} is compared to the identity via the normalized integral $2/L^2 \int |h_i^{-1}(t) - t| dt$, where L is the domain length. The normalizing constant $L^2/2$ is the theoretical upper limit deviation for a monotone, domain-preserving warp that maps all timepoints to the first or last timepoint, so values range from 0 (identity warp, no time deformation) to 1 (maximal crazy warping). Values above ≈ 0.3 may suggest aggressive warping that could warrant inspection.

Warp slopes (deciles of per-curve min and max dh^{-1}/dt): a slope of 1 of the warp corresponds to no local time deformation (identity). Slopes > 1 indicate local time dilation (the warped curve is "stretched" relative to the template), slopes < 1 indicate local time compression, so slopes near 0 or very large slopes indicate extreme local deformation. For affine shift warps, all slopes are exactly 1.

Domain coverage loss (only printed if any loss occurs): the fraction of the original domain range that is lost per curve after alignment, computed as $1 - \text{range}(\text{aligned_arg}) / \text{range}(\text{original_arg})$. This is only relevant for affine (non-domain-preserving) warps where alignment can shift parts of curves outside the original domain. Domain-preserving methods (`srvf`, `cc`, `landmark`) always have zero domain loss.

Accessors

- `tf_aligned(x)`: extract the registered/aligned curves (tfd vector).
- `tf_inv_warps(x)`: extract the estimated inverse warping functions $h_i^{-1}(t)$ that map observed time to aligned time (tfd vector). Use `tf_invert()` on the result to obtain forward warps if needed.
- `tf_template(x)`: extract the template function (tf vector of length 1).
- `tf_rotations(x)`: extract the per-curve rotation matrices from a `tf_shape_registration` object.
- `tf_scales(x)`: extract the per-curve scale factors from a `tf_shape_registration` object. Each factor is relative to the template (template SRVF norm divided by the curve's SRVF norm); see `tf_register_shape()`.

Author(s)

Fabian Scheipl, Claude Opus 4.6

See Also

Other registration functions: `tf_align()`, `tf_estimate_warps()`, `tf_landmarks_extrema()`, `tf_register()`, `tf_register_shape()`, `tf_warp()`

Examples

```
reg <- tf_register(pinch[1:5], method = "affine", type = "shift_scale")
reg
summary(reg)
plot(reg)
```

tf_rgp

Gaussian Process random generator

Description

Generates n realizations of a zero-mean Gaussian process. The function also accepts user-defined covariance functions (without "nugget" effect, see `cov`). The implemented defaults with scale parameter ϕ , order o and nugget effect variance σ^2 are:

- *squared exponential*: $Cov(x(t), x(t')) = \exp(-(t - t')^2)/\phi) + \sigma^2\delta_t(t')$.
- *Wiener process*: $Cov(x(t), x(t')) = \min(t', t)/\phi + \sigma^2\delta_t(t')$,
- *Matérn process*: $Cov(x(t), x(t')) = \frac{2^{1-o}}{\Gamma(o)} \left(\frac{\sqrt{2o}|t-t'|}{\phi}\right)^o \text{Bessel}_o\left(\frac{\sqrt{2o}|t-t'|}{s}\right) + \sigma^2\delta_t(t')$
- *Brownian Bridge process* for $t, t' \in [a, b]$: $Cov(x(t), x(t')) = \frac{(b-\max(s,t))(\min(s,t)-a)}{\phi(b-a)} + \sigma^2\delta_t(t')$

Usage

```
tf_rgp(
  n,
  arg = 51L,
  cov = c("squareexp", "wiener", "matern", "brown_bridge"),
  scale = diff(domain)/10,
  nugget = scale/200,
  order = 1.5,
  domain = NULL
)
```

Arguments

n	how many realizations to draw.
arg	vector of evaluation points (arg of the return object). Defaults to (0, 0.02, 0.04, ..., 1). If given as a single integer (don't forget the L...), creates a regular grid of that length over (0,1). If given as a n-long list of vectors, irregular functional data are created.
cov	type of covariance function to use. Implemented defaults are "squareexp", "wiener", "matern", see description. Can also be any vectorized function returning $Cov(x(t), x(t'))$ without nugget effect for pairs of inputs t and t'.
scale	scale parameter (see description). Defaults to the width of the domain divided by 10.
nugget	nugget effect for additional white noise / unstructured variability. Defaults to scale/200 (so: very little white noise).
order	order of the Matérn covariance (if used, must be >0), defaults to 1.5. The higher, the smoother the process. Evaluation of the covariance function becomes numerically unstable for large (>20) order, use "squareexp".
domain	of the generated functions. If not provided, the range of the supplied arg values.

Value

an tfd-vector of length n.

See Also

Other tidyfun RNG functions: [tf_jiggle\(\)](#)

Examples

```
(x1 <- tf_rgp(10, cov = "squareexp", nugget = 0))
tf_rgp(2, arg = list(sort(runif(25)), sort(runif(34))))
```

tf_smooth

Simple smoothing of tf objects

Description

Apply running means or medians, lowess or Savitzky-Golay filtering to smooth functional data. This does nothing for tfb-objects, which should be smoothed by using a smaller basis / stronger penalty.

Usage

```
tf_smooth(x, ...)

## S3 method for class 'tfb'
tf_smooth(x, verbose = TRUE, ...)

## S3 method for class 'tfd'
tf_smooth(
  x,
  method = c("lowess", "rollmean", "rollmedian", "savgol"),
  verbose = TRUE,
  ...
)
```

Arguments

x	a tf object containing functional data.
...	arguments for the respective method. See details.
verbose	give lots of diagnostic messages? Defaults to TRUE.
method	one of "lowess" (see stats::lowess()), "rollmean", "rollmedian" (see zoo::rollmean()) or "savgol" (a Savitzky-Golay filter, see savgol()).

Details

`tf_smooth.tfd` overrides/automatically sets some defaults of the used methods:

- `lowess` uses a span parameter of $f = 0.15$ (instead of 0.75) by default.
- `rollmean/median` use a window size of $k = \lceil \text{number of grid points} / 20 \rceil$ (i.e., the nearest odd integer to that) and sets `fill = "extend"` (i.e., constant extrapolation to replace missing values at the extremes of the domain) by default. Use `fill = NA` for zoo's default behavior of shortening the smoothed series.
- `savgol` uses a window size of $k = \lceil \text{number of grid points} / 10 \rceil$ (i.e., the nearest odd integer to that).

Value

a smoothed version of the input. For some methods/options, the smoothed functions may be shorter than the original ones (at both ends).

Examples

```
library(zoo)
f <- tf_sparsify(tf_jiggle(tf_rgp(4, 201, nugget = 0.05)))
f_lowess <- tf_smooth(f, "lowess")
# these methods ignore the distances between arg-values:
f_mean <- tf_smooth(f, "rollmean")
f_median <- tf_smooth(f, "rollmedian", k = 31)
f_sg <- tf_smooth(f, "savgol", fl = 31)
```

```

layout(t(1:4))
plot(f, points = FALSE, main = "original")
plot(f_lowess,
     points = FALSE, col = "blue", main = "lowess (default,\n span 0.9 in red)"
)
lines(tf_smooth(f, "lowess", f = 0.9), col = "red", alpha = 0.2)
plot(f_mean,
     points = FALSE, col = "blue", main = "rolling means &\n medians (red)"
)
lines(f_median, col = "red", alpha = 0.2) # note constant extrapolation at both ends!
plot(f, points = FALSE, main = "original and\n savgol (red)")
lines(f_sg, col = "red")

```

tf_split

Split / Combine functional fragments

Description

tf_split separates each function into a vector of functions defined on a sub-interval of its domain, either with overlap at the cut points or without.

tf_combine joins functional fragments together to create longer (or more densely evaluated) functions.

Usage

```
tf_split(x, splits, include = c("both", "left", "right"))
```

```
tf_combine(..., strict = FALSE)
```

Arguments

x	a tf object.
splits	numeric vector containing arg-values at which to split.
include	which of the end points defined by splits to include in each of the resulting split functions. Defaults to "both", other options are "left" or "right". See examples.
...	tf-objects of identical lengths to combine
strict	only combine functions whose argument ranges do not overlap, are given in the correct order & contain no duplicate values at identical arguments? defaults to FALSE. If strict == FALSE, only the first function values at duplicate locations are used, the rest are discarded (with a warning).

Value

for tf_split: a list of tf objects.

for tf_combine: a tfd with the combined subfunctions on the union of the input tf_arg-values

Examples

```
x <- tfd(1:100, arg = 1:100)
tf_split(x, splits = c(20, 80))
tf_split(x, splits = c(20, 80), include = "left")
tf_split(x, splits = c(20, 80), include = "right")
x <- tf_rgp(5)
tfs <- tf_split(x, splits = c(.2, .6))
x2 <- tf_combine(tfs[[1]], tfs[[2]], tfs[[3]])
# tf_combine(tfs[[1]], tfs[[2]], tfs[[3]], strict = TRUE) # errors out - duplicate values!
all.equal(x, x2)
# combine works for different input types:
tfs2_sparse <- tf_sparsify(tfs[[2]])
tfs3_spline <- tfb(tfs[[3]])
tf_combine(tfs[[1]], tfs2_sparse, tfs3_spline)
# combine(..., strict = FALSE) can be used to coalesce different measurements
# of the same process over different grids:
x1 <- tfd(x, arg = tf_arg(x)[seq(1, 51, by = 2)])
x2 <- tfd(x, arg = tf_arg(x)[seq(2, 50, by = 2)])
tf_combine(x2, x1, strict = FALSE) == x

plot(tf_combine(x2, x1, strict = FALSE))
points(x1, col = "blue", pch = "x")
points(x2, col = "red", pch = "o")
```

tf_warp

Elastic Deformation: warp and align tf vectors

Description

These functions stretch and/or compress regions of the domain of functional data:

- `tf_warp()` applies warping functions to aligned (registered) functional data to recover the original unregistered curves: $x(s) \rightarrow x(h(s)) = x(t)$.
- `tf_align()` applies the *inverse* warping function to unregistered data to obtain aligned (registered) functions: $x(t) \rightarrow x(h^{-1}(t)) = x(s)$.

Usage

```
tf_warp(x, warp, ...)

## S3 method for class 'tfd'
tf_warp(x, warp, ..., keep_new_arg = FALSE)

## S3 method for class 'tfb'
tf_warp(x, warp, ...)
```

Arguments

x	tf vector of functions. For <code>tf_warp()</code> , these should be registered/aligned functions and unaligned functions for <code>tf_align()</code> .
warp	tf vector of warping functions used for transformation. See Details.
...	additional arguments passed to <code>tfd()</code> .
keep_new_arg	keep new arg values after (un)warping or return tfd vector on arg values of the input (default FALSE is the latter)? See Details.

Details

These functions will work best with functions evaluated on suitably dense and regular grids.

Warping functions $h(s) = t$ are strictly monotone increasing (no time travel backwards or infinite time dilation) with identical domain and co-domain: $h : T \rightarrow T$. Their input is the aligned "system" time s , their output is the unaligned "observed" time t .

By default (`keep_new_arg = FALSE`), the `tfd` methods will return function objects re-evaluated on the same grids as the original inputs, which will typically incur some additional interpolation error because (un)warping changes the underlying grids, which are then changed back. Set to `TRUE` to avoid. This option is not available for `tfb`-objects.

Value

- `tf_warp()`: the warped tf vector (un-registered functions)
- `tf_align()`: the aligned tf vector (registered functions)

Author(s)

Maximilian Muecke, Fabian Scheipl, Claude Opus 4.6

See Also

Other registration functions: `tf_align()`, `tf_estimate_warps()`, `tf_landmarks_extrema()`, `tf_register()`, `tf_register_shape()`, `tf_registration`

Examples

```
# generate "template" function shape on [0, 1]:
set.seed(1351)
template <- tf_rgp(1, arg = 201L, nugget = 0)
# generate random warping functions (strictly monotone inc., [0, 1] -> [0, 1]):
warp <- {
  tmp <- tf_rgp(5)
  tmp <- exp(tmp - mean(tmp)) # centered at identity warping
  tf_integrate(tmp, definite = FALSE) / tf_integrate(tmp)
}
x <- tf_warp(rep(1, 5) * template, warp)
layout(t(1:3))
plot(template); plot(warp, col = 1:5); plot(x, col = 1:5)
# register the functions:
```

```

if (requireNamespace("fdaSrvf", quietly = TRUE)) {
  reg <- tf_register(x)
} else {
  reg <- tf_register(x, method = "affine", type = "shift_scale")
}
layout(t(1:3))
plot(x, col = 1:5)
plot(tf_inv_warps(reg), col = 1:5); lines(tf_invert(warp), lty = 3, lwd = 1.5, col = 1:5)
plot(tf_aligned(reg), col = 1:5, points = FALSE); lines(template, lty = 2)

```

tf_where

Find out where functional data fulfills certain conditions.

Description

tf_where allows to define a logical expression about the function values and returns the argument values for which that condition is true.

tf_anywhere is syntactic sugar for tf_where with return = "any" to get a logical flag for each function if the condition is TRUE *anywhere*, see below.

Usage

```

tf_where(
  f,
  cond,
  return = c("all", "first", "last", "range", "any"),
  arg = tf_arg(f)
)

tf_anywhere(f, cond, arg = tf_arg(f))

```

Arguments

f	a tf object.
cond	a logical expression about value (and/or arg) that defines a condition about the functions, see examples and details.
return	for each entry in f, tf_where either returns <i>all</i> arg for which cond is true, the <i>first</i> , the <i>last</i> or their <i>range</i> or logical flags whether the functions fulfill the condition <i>anywhere</i> . For "range", note that cond may not be true for all arg values in this range, though, this is not checked.
arg	optional arg-values on which to evaluate f and check cond, defaults to tf_arg(f).

Details

Entries in f that do not fulfill cond anywhere yield numeric(0).

cond is evaluated as a `base::subset()`-statement on a data.frame containing a single entry in f

with columns `arg` and `value`, so most of the usual `dplyr` tricks are available as well, see examples. Any condition evaluates to NA on NA-entries in `f`.

For vector-valued (`tf_mv`) input, the data.frame on which `cond` is evaluated has one column per component, named like the components, instead of a single value column: conditions are *joint* conditions across components, e.g. `tf_where(f, x > 0 & y < 1)` or `tf_where(f, sqrt(x^2 + y^2) > 1)` for components "x" and "y". To apply a condition to a single component, extract it first: `tf_where(f[, component = "x"], value > 0)`. All components have to be observed on a common grid – use `tf_interpolate()` to align them first, or supply `arg` explicitly.

Value

depends on return:

- `return = "any"`, i.e, anywhere: a logical vector of the same length as `f`.
- `return = "all"`: a list of vectors of the same length as `f`, with empty vectors for the functions that never fulfill the condition.
- `return = "range"`: a data frame with columns "begin" and "end".
- else, a numeric vector of the same length as `f` with NA for entries of `f` that nowhere fulfill the condition.

Examples

```
lin <- 1:4 * tfd(seq(-1, 1, length.out = 11), seq(-1, 1, length.out = 11))
tf_where(lin, value %inr% c(-1, 0.5))
tf_where(lin, value %inr% c(-1, 0.5), "range")
a <- 1
tf_where(lin, value > a, "first")
tf_where(lin, value < a, "last")
tf_where(lin, value > 2, "any")
tf_anywhere(lin, value > 2)

set.seed(4353)
f <- tf_rgp(5, 11)
plot(f, pch = as.character(1:5), points = TRUE)
tf_where(f, value == max(value))
# where is the function increasing/decreasing?
tf_where(f, value > dplyr::lag(value, 1, value[1]))
tf_where(f, value < dplyr::lead(value, 1, tail(value, 1)))
# where are the (interior) extreme points (sign changes of `diff(value)`)?
tf_where(
  f,
  sign(c(diff(value)[1], diff(value))) !=
  sign(c(diff(value), tail(diff(value), 1)))
)
# where in its second half is the function positive?
tf_where(f, arg > 0.5 & value > 0)
# does the function ever exceed?
tf_anywhere(f, value > 1)

# vector-valued input: conditions refer to components by name
```

```
fm <- tfd_mv(list(x = tf_rgp(3, 11L), y = tf_rgp(3, 11L)))
tf_where(fm, x > 0 & y < 0)
tf_where(fm, sqrt(x^2 + y^2) > 1, "first")
tf_anywhere(fm, x > y)
```

tf_zoom

*Functions to zoom in/out on functions***Description**

These are used to redefine or restrict the domain of tf objects.

Usage

```
tf_zoom(f, begin, end, ...)

## S3 method for class 'tfd'
tf_zoom(f, begin = tf_domain(f)[1], end = tf_domain(f)[2], ...)

## S3 method for class 'tfb'
tf_zoom(f, begin = tf_domain(f)[1], end = tf_domain(f)[2], ...)

## S3 method for class 'tfb_fpc'
tf_zoom(f, begin = tf_domain(f)[1], end = tf_domain(f)[2], ...)
```

Arguments

f	a tf-object.
begin	numeric vector of length 1 or length(f). Defaults to the lower limit of the domain of f.
end	numeric vector of length 1 or length(f). Defaults to the upper limit of the domain of f.
...	not used

Value

an object like f on a new domain (potentially). Note that regular functional data and functions in basis representation will be turned into irregular tfd-objects if begin or end are not scalar.

See Also

Other tidyfun utility functions: [ensure_list\(\)](#), [in_range\(\)](#), [tf_arg\(\)](#), [unique_id\(\)](#)

Examples

```
x <- tf_rgp(10)
plot(x)
tf_zoom(x, 0.5, 0.9)
tf_zoom(x, 0.5, 0.9) |> lines(col = "red")
tf_zoom(x, seq(0, 0.5, length.out = 10), seq(0.5, 1, length.out = 10)) |>
  lines(col = "blue", lty = 3)
```

unique_id	<i>Make syntactically valid unique names</i>
-----------	--

Description

Coerces `x` to character and returns syntactically valid, unique identifiers. Empty strings are replaced with "NA" before deduplication. If `x` already has no duplicates it is returned unchanged.

Usage

```
unique_id(x)
```

Arguments

`x` any input that can be coerced to character.

Value

A character vector of unique, syntactically valid names of the same length as `x`.

See Also

Other tidyfun utility functions: [ensure_list\(\)](#), [in_range\(\)](#), [tf_arg\(\)](#), [tf_zoom\(\)](#)

Examples

```
unique_id(c("a", "a", "b"))
unique_id(c(1, 1, 2))
```

vctrs	<i>vctrs methods for tf objects</i>
-------	-------------------------------------

Description

These functions are the extensions that allow tf vectors to work with vctrs.

Usage

```
## S3 method for class 'tfd_mv.tfd_mv'
vec_ptype2(x, y, ...)

## S3 method for class 'tfb_mv.tfb_mv'
vec_ptype2(x, y, ...)

## S3 method for class 'tfd_mv.tfb_mv'
vec_ptype2(x, y, ...)

## S3 method for class 'tfb_mv.tfd_mv'
vec_ptype2(x, y, ...)

## S3 method for class 'tfd_mv.tfd_mv'
vec_cast(x, to, ...)

## S3 method for class 'tfb_mv.tfb_mv'
vec_cast(x, to, ...)

## S3 method for class 'tfd_mv.tfb_mv'
vec_cast(x, to, ...)

## S3 method for class 'tfb_mv.tfd_mv'
vec_cast(x, to, ...)

## S3 method for class 'tfd_reg.tfd_reg'
vec_cast(x, to, ...)

## S3 method for class 'tfd_reg.tfd_irreg'
vec_cast(x, to, ...)

## S3 method for class 'tfd_reg.tfb_spline'
vec_cast(x, to, ...)

## S3 method for class 'tfd_reg.tfb_fpc'
vec_cast(x, to, ...)

## S3 method for class 'tfd_irreg.tfd_reg'
vec_cast(x, to, ...)
```

```
## S3 method for class 'tfd_irreg.tfd_irreg'
vec_cast(x, to, ...)

## S3 method for class 'tfd_irreg.tfb_spline'
vec_cast(x, to, ...)

## S3 method for class 'tfd_irreg.tfb_fpc'
vec_cast(x, to, ...)

## S3 method for class 'tfb_spline.tfb_spline'
vec_cast(x, to, ...)

## S3 method for class 'tfb_spline.tfb_fpc'
vec_cast(x, to, ...)

## S3 method for class 'tfb_fpc.tfb_spline'
vec_cast(x, to, ...)

## S3 method for class 'tfb_fpc.tfb_fpc'
vec_cast(x, to, ...)

## S3 method for class 'tfb_spline.tfd_reg'
vec_cast(x, to, ...)

## S3 method for class 'tfb_spline.tfd_irreg'
vec_cast(x, to, ...)

## S3 method for class 'tfb_fpc.tfd_reg'
vec_cast(x, to, ...)

## S3 method for class 'tfb_fpc.tfd_irreg'
vec_cast(x, to, ...)

## S3 method for class 'tfd_reg.tfd_reg'
vec_ptype2(x, y, ...)

## S3 method for class 'tfd_reg.tfd_irreg'
vec_ptype2(x, y, ...)

## S3 method for class 'tfd_reg.tfb_spline'
vec_ptype2(x, y, ...)

## S3 method for class 'tfd_reg.tfb_fpc'
vec_ptype2(x, y, ...)

## S3 method for class 'tfd_irreg.tfd_reg'
vec_ptype2(x, y, ...)
```

```
## S3 method for class 'tfd_irreg.tfd_irreg'
vec_ptype2(x, y, ...)

## S3 method for class 'tfd_irreg.tfb_spline'
vec_ptype2(x, y, ...)

## S3 method for class 'tfd_irreg.tfb_fpc'
vec_ptype2(x, y, ...)

## S3 method for class 'tfb_spline.tfb_spline'
vec_ptype2(x, y, ...)

## S3 method for class 'tfb_spline.tfb_fpc'
vec_ptype2(x, y, ...)

## S3 method for class 'tfb_spline.tfd_reg'
vec_ptype2(x, y, ...)

## S3 method for class 'tfb_spline.tfd_irreg'
vec_ptype2(x, y, ...)

## S3 method for class 'tfb_fpc.tfb_spline'
vec_ptype2(x, y, ...)

## S3 method for class 'tfb_fpc.tfb_fpc'
vec_ptype2(x, y, ...)

## S3 method for class 'tfb_fpc.tfd_reg'
vec_ptype2(x, y, ...)

## S3 method for class 'tfb_fpc.tfd_irreg'
vec_ptype2(x, y, ...)
```

Arguments

<code>x</code>	Vectors to cast.
<code>y</code>	vectors to cast.
<code>...</code>	For <code>vec_cast_common()</code> , vectors to cast. For <code>vec_cast()</code> , <code>vec_cast_default()</code> , and <code>vec_restore()</code> , these dots are only for future extensions and should be empty.
<code>to</code>	Type to cast to. If NULL, <code>x</code> will be returned as is.

Details

Notes on `vec_cast`: Use `tf_rebase()` to change the representations of `tf`-vectors, these methods are only for internal use – automatic/implicit casting of `tf` objects is tricky because it's hard to determine automatically whether such an operation would lose precision (different bases with

different expressivity? different argument grids?), and it's not generally clear which instances of which tf-subclasses should be considered the "richer" objects. Rules for casting:

- If the casted object's domain would not contain the entire original domain, no casting is possible (would lose data).
- Every cast that evaluates (basis) functions on different arg values is a *lossy* cast, since it might lose precision (`vctrs::maybe_lossy_cast`).
- As long as the casted object's domain contains the entire original domain:
 - every `tfd_reg`, `tfd_irreg` or `tfb` can always be cast into an equivalent `tfd_irreg` (which may also change its evaluator and domain).
 - every `tfd_reg` can always be cast to `tfd_reg` (which may change its evaluator and domain)
 - every `tfb` can be cast *losslessly* to `tfd` (regular or irregular, note it's lossless only on the *original* arg-grid)
- Any cast of a `tfd` into `tfb` is potentially *lossy* (because we don't know how expressive the chosen basis is)
- Only `tfb` with identical bases and domains can be cast into one another *losslessly*
- Casts deliberately **keep the source's arg-grid**: they *re-express* functions in the target's representation rather than re-evaluate them on the target's grid, so a cast result need not be identical to the target prototype (a deliberate deviation from the strict `vctr`s cast invariant that avoids silently coarsening data during type unification).

Value

for `vec_cast`: the casted tf-vector, for `vec_ptype2`: the common prototype

See Also

[vctrs::vec_cast\(\)](#), [vctrs::vec_ptype2\(\)](#)

Examples

```
set.seed(1)
x <- tf_rgp(3)
xi <- tf_sparsify(x)
# different tf subtypes combine to their common type:
c(x, xi)
vctrs::vec_ptype_full(vctrs::vec_c(x, xi))
```

[.tf_mv

Accessing, evaluating, subsetting and subassigning tf vectors

Description

These functions access, subset, replace and evaluate tf objects. For more information on creating tf objects and converting them to/from list, data.frame or matrix, see [tfd\(\)](#) and [tfb\(\)](#). See details.

Usage

```
## S3 method for class 'tf_mv'
x[i, j, component = NULL, interpolate = TRUE, matrix = TRUE]

## S3 replacement method for class 'tf_mv'
x[i] <- value

## S3 method for class 'tf'
x[i, j, interpolate = TRUE, matrix = TRUE]

## S3 replacement method for class 'tf'
x[i] <- value
```

Arguments

<code>x</code>	an <code>tf</code> .
<code>i</code>	index of the observations (integerish, character or logical, usual R rules apply). Can also be a two-column matrix for extracting specific (function, arg-value) pairs: the first column gives the function indices, the second column gives the arg values at which to evaluate each function. Returns a numeric vector in that case. <code>j</code> must not be provided when <code>i</code> is a matrix.
<code>j</code>	The arg used to evaluate the functions. A (list of) numeric vectors. <i>NOT</i> interpreted as a column number but as the argument value of the respective functional datum. If <code>j</code> is missing but <code>matrix</code> is explicitly given, <code>j</code> defaults to <code>tf_arg(x)</code> .
<code>component</code>	for <code>tf_mv</code> objects only: optionally restrict evaluation / extraction to a subset of the output dimensions, given by name or integer index. A single name/index drops to the univariate component (a <code>tfd</code> or <code>tfb</code>); a vector of length > 1 returns a sub- <code>tf_mv</code> containing just those components. <code>NULL</code> (default) keeps all <code>d</code> components.
<code>interpolate</code>	should functions be evaluated (i.e., inter-/extrapolated) for values in <code>arg</code> for which no original data is available? Only relevant for the raw data class <code>tfd</code> , for which it defaults to <code>TRUE</code> . Basis-represented <code>tfb</code> are always "interpolated".
<code>matrix</code>	should the result be returned as a matrix or as a list of data.frames? If <code>TRUE</code> , <code>j</code> has to be a (list of a) single vector of <code>arg</code> . See return value.
<code>value</code>	<code>tf</code> object for subassignment. This is typed more strictly than concatenation: subassignment only happens if the common type of <code>value</code> and <code>x</code> is the same as the type of <code>x</code> , so subassignment never changes the type of <code>x</code> but may do a potentially lossy cast of <code>value</code> to the type of <code>x</code> (with a warning).

Details

Note that these break certain (terrible) R conventions for vector-like objects:

- no argument recycling,
- no indexing with `NA`,

- no indexing with names not present in `x`,
- no indexing with integers $> \text{length}(x)$

All of the above will trigger errors.

Value

If `i` is a two-column matrix, a numeric vector of pointwise evaluations (one per row of `i`).

If `j` is missing (and `i` is not a matrix), a subset of the functions in `x` as given by `i`.

If `j` is given and `matrix == TRUE`, a numeric matrix of function evaluations in which each row represents one function and each column represents one `argval` as given in argument `j`, with an attribute `arg=j` and row- and column-names derived from `x[i]` and `j`.

If `j` is given and `matrix == FALSE`, a list of `tbl_dfs` with columns `arg = j` and `value = evaluations` at `j` for each observation in `i`.

Examples

```
x <- 1:3 * tfd(data = 0:10, arg = 0:10)
plot(x)
# this operator's 2nd argument is quite overloaded -- you can:
# 1. simply extract elements from the vector if no second arg is given:
x[1]
x[c(TRUE, FALSE, FALSE)]
x[-(2:3)]
# 2. use the second argument and optional additional arguments to
#   extract specific function evaluations in a number of formats:
x[1:2, c(4.5, 9)] # returns a matrix of function evaluations
x[1:2, c(4.5, 9), interpolate = FALSE] # NA for arg-values not in the original data
x[-3, seq(1, 9, by = 2), matrix = FALSE] # list of data.frames for each function
# 3. use a 2-column matrix to extract specific (function, arg) pairs:
x[cbind(1:3, c(0, 5, 10))] # one value per function
# 4. use matrix= with a missing j to evaluate on the default arg grid:
x[1:2, , matrix = FALSE] # same as x[1:2, tf_arg(x), matrix = FALSE]
# in order to evaluate a set of observed functions on a new grid and
# save them as a functional data vector again, use `tfd` or `tfb` instead:
tfd(x, arg = seq(0, 10, by = 0.01))
```

Index

- `!=.tfb(tfgroupgenerics)`, 38
- `!=.tfd(tfgroupgenerics)`, 38
- * **datasets**
 - `gait`, 12
 - `growth`, 13
 - `pinch`, 15
- * **registration functions**
 - `tf_align`, 42
 - `tf_estimate_warps`, 53
 - `tf_landmarks_extrema`, 63
 - `tf_register`, 70
 - `tf_register_shape`, 73
 - `tf_registration`, 74
 - `tf_warp`, 81
- * **tf_mv-class**
 - `plot.tf_mv`, 18
 - `tf_arclength`, 45
 - `tf_geom`, 57
 - `tf_mv_methods`, 66
 - `tfb_mfpc`, 25
 - `tfb_mv`, 27
 - `tfd_mv`, 36
- * **tfb-class**
 - `fpc_wsvd`, 8
 - `tfb`, 21
 - `tfb_fpc`, 22
 - `tfb_spline`, 29
- * **tfb_fpc-class**
 - `fpc_wsvd`, 8
 - `tfb_fpc`, 22
 - `tfb_mfpc`, 25
- * **tfb_spline-class**
 - `tfb_spline`, 29
- * **tfd-class**
 - `tfd`, 33
- * **tidyfun RNG functions**
 - `tf_jiggle`, 62
 - `tf_rgp`, 77
- * **tidyfun bracket-operator**
 - `[.tf_mv`, 90
- * **tidyfun calculus functions**
 - `tf_derive`, 51
 - `tf_integrate`, 59
- * **tidyfun compute functions**
 - `tfgroupgenerics`, 38
- * **tidyfun compute**
 - `tfgroupgenerics`, 38
- * **tidyfun converters**
 - `as.data.frame.tf`, 3
 - `converters-mv`, 5
- * **tidyfun developer tools**
 - `prep_plotting_arg`, 19
- * **tidyfun inter/extrapolation functions**
 - `tf_approx_linear`, 43
 - `tf_evaluate`, 55
 - `tf_interpolate`, 60
- * **tidyfun nonparametric smoothers**
 - `tf_smooth`, 78
- * **tidyfun ordering and ranking functions**
 - `tf_depth`, 49
 - `tf_minmax`, 65
 - `tf_order`, 67
- * **tidyfun print**
 - `print.tf`, 19
- * **tidyfun query-functions**
 - `tf_where`, 83
- * **tidyfun setters**
 - `tf_interpolate`, 60
- * **tidyfun summary functions**
 - `fivenum`, 7
 - `functionwise`, 9
 - `median.tf_mv`, 14
 - `tfsummaries`, 40
- * **tidyfun utility functions**
 - `ensure_list`, 6
 - `in_range`, 14
 - `tf_arg`, 46
 - `tf_zoom`, 85

- unique_id, 86
- * **tidyfun vctrs**
 - vctrs, 87
- * **tidyfun visualization**
 - plot.tf, 16
- ==.tfb(tfgroupgenerics), 38
- ==.tfd(tfgroupgenerics), 38
- [.tf([.tf_mv), 90
- [.tf_mv, 90
- [.tf_registration(tf_registration), 74
- [.tf_shape_registration
 - (tf_registration), 74
- [<-.tf([.tf_mv), 90
- [<-.tf_mv([.tf_mv), 90
- %inr%(in_range), 14
- as.data.frame.tf, 3
- as.data.frame.tf(), 6
- as.data.frame.tf_mv(converters-mv), 5
- as.function.tf(as.data.frame.tf), 3
- as.matrix.tf, 5
- as.matrix.tf(as.data.frame.tf), 3
- as.matrix.tf(), 6
- as.matrix.tf_mv(converters-mv), 5
- as.tfb(tfb), 21
- as.tfb_mv(tfb_mv), 27
- as.tfd(tfd), 33
- as.tfd_irreg(tfd), 33
- as.tfd_mv(tfd_mv), 36
- as.tfd_mv(), 26
- base::rank(), 68
- base::sort(), 68
- base::subset(), 83
- cli::spark_bar(), 20
- coef.tfb(tf_arg), 46
- converters-mv, 5
- cummax.tfb(tfgroupgenerics), 38
- cummax.tfd(tfgroupgenerics), 38
- cummin.tfb(tfgroupgenerics), 38
- cummin.tfd(tfgroupgenerics), 38
- cumprod.tfb(tfgroupgenerics), 38
- cumprod.tfd(tfgroupgenerics), 38
- cumsum.tfb(tfgroupgenerics), 38
- cumsum.tfd(tfgroupgenerics), 38
- ensure_list, 6
- ensure_list(), 14, 48, 85, 86
- fdasrvf::time_warping(), 53, 54
- fivenum, 7
- fivenum(), 11, 15, 42
- format.tf(print.tf), 19
- format.tf(), 20
- fpc_wsvd, 8
- fpc_wsvd(), 22–24, 26, 27, 33
- functionwise, 8, 9, 15, 42
- gait, 12
- getOption, 20
- graphics::matplot(), 17
- grDevices::rgb(), 17
- growth, 13
- image(), 17
- in_range, 14
- in_range(), 7, 48, 85, 86
- is.na.tf(tf_arg), 46
- is.na.tfd_irreg(tf_arg), 46
- is_irreg(tf_arg), 46
- is_reg(tf_arg), 46
- is_tf(tf_arg), 46
- is_tf_1d(tf_arg), 46
- is_tf_1d(), 37
- is_tf_mv(tf_arg), 46
- is_tfb(tf_arg), 46
- is_tfb_fpc(tf_arg), 46
- is_tfb_mfpc(tfb_mfpc), 25
- is_tfb_mv(tf_arg), 46
- is_tfb_spline(tf_arg), 46
- is_tfd(tf_arg), 46
- is_tfd_irreg(tf_arg), 46
- is_tfd_mv(tf_arg), 46
- is_tfd_reg(tf_arg), 46
- length.tf_registration
 - (tf_registration), 74
- lines.tf(plot.tf), 16
- lines.tf_mv(plot.tf_mv), 18
- Math.tfb(tfgroupgenerics), 38
- Math.tfd(tfgroupgenerics), 38
- max.tf(tf_minmax), 65
- max.tf(), 69
- mean(), 66
- mean.tf(tfsummaries), 40
- median(), 66
- median.tf(tfsummaries), 40

median.tf(), 14
 median.tf_mv, 14
 median.tf_mv(), 8, 11, 42
 mgcv::family.mgcv(), 32
 mgcv::gam(), 32
 mgcv::gam.fit(), 31
 mgcv::magic(), 31, 32
 mgcv::s(), 31, 32
 mgcv::smooth.terms(), 32, 33
 min.tf(tf_minmax), 65
 min.tf(), 69

 pillar::glimpse(), 21
 pinch, 15
 plot.tf, 16
 plot.tf(), 18
 plot.tf_mv, 18
 plot.tf_mv(), 27, 28, 37, 46, 58, 66
 plot.tf_registration(tf_registration), 74
 points.tf(plot.tf), 16
 points.tf_mv(plot.tf_mv), 18
 prep_plotting_arg, 19
 print.summary.tf_registration
 (tf_registration), 74
 print.summary.tf_shape_registration
 (tf_registration), 74
 print.tf, 19
 print.tf_registration
 (tf_registration), 74
 print.tf_shape_registration
 (tf_registration), 74
 print.tfb(print.tf), 19
 print.tfd_irreg(print.tf), 19
 print.tfd_reg(print.tf), 19
 purrr::as_mapper(), 10

 range.tf(tf_minmax), 65
 rank(tf_order), 67
 rank.tf(), 65
 rev.tf(tf_arg), 46
 rev.tf_mv(tf_arg), 46

 savgol(), 79
 sd(tfsummaries), 40
 signif, 20
 sort.tf(tf_order), 67
 stats::fivenum(), 7
 stats::lowess(), 79

 Summary.tf(tfgroupgenerics), 38
 summary.tf(tfsummaries), 40
 summary.tf_mv(tfsummaries), 40
 summary.tf_registration
 (tf_registration), 74
 summary.tf_shape_registration
 (tf_registration), 74

 tf_align, 42
 tf_align(), 54, 55, 64, 66, 72, 74, 77, 82
 tf_aligned(tf_registration), 74
 tf_aligned(), 70, 71, 74
 tf_anywhere(tf_where), 83
 tf_approx_fill_extend
 (tf_approx_linear), 43
 tf_approx_fill_extend(), 60
 tf_approx_linear, 43
 tf_approx_linear(), 56, 61
 tf_approx_locf(tf_approx_linear), 43
 tf_approx_nocb(tf_approx_linear), 43
 tf_approx_none(tf_approx_linear), 43
 tf_approx_spline(tf_approx_linear), 43
 tf_arclength, 45
 tf_arclength(), 18, 27, 28, 37, 58, 66
 tf_arg, 46
 tf_arg(), 7, 14, 85, 86
 tf_arg(x), 91
 tf_arg<-(tf_arg), 46
 tf_basis(tf_arg), 46
 tf_combine(tf_split), 80
 tf_component(tf_mv_methods), 66
 tf_component<-(tf_mv_methods), 66
 tf_components(tf_mv_methods), 66
 tf_components(), 37
 tf_count(tf_arg), 46
 tf_crosscor(functionwise), 9
 tf_crosscov(functionwise), 9
 tf_depth, 49
 tf_depth(), 7, 14, 15, 41, 42, 65, 67–69
 tf_derive, 51
 tf_derive(), 46, 60, 66
 tf_distance(tf_geom), 57
 tf_domain(tf_arg), 46
 tf_domain<-(tf_arg), 46
 tf_estimate_warps, 53
 tf_estimate_warps(), 43, 64, 66, 70–72, 74, 77, 82
 tf_evaluate, 55
 tf_evaluate(), 6, 43, 44, 61

- tf_evaluations (tf_arg), 46
- tf_evaluator (tf_arg), 46
- tf_evaluator(), 9
- tf_evaluator<- (tf_arg), 46
- tf_fmax (functionwise), 9
- tf_fmax(), 11
- tf_fmean (functionwise), 9
- tf_fmedian (functionwise), 9
- tf_fmin (functionwise), 9
- tf_fmin(), 11
- tf_frange (functionwise), 9
- tf_fsd (functionwise), 9
- tf_fvar (functionwise), 9
- tf_fwise (functionwise), 9
- tf_fwise(), 40, 42
- tf_geom, 18, 27, 28, 37, 46, 57, 66
- tf_inner (tf_geom), 57
- tf_integrate, 59
- tf_integrate(), 45, 46, 52, 66
- tf_interpolate, 60
- tf_interpolate(), 44, 56, 84
- tf_inv_warps (tf_registration), 74
- tf_inv_warps(), 70, 71, 74
- tf_invert, 62
- tf_invert(), 54, 76
- tf_jiggle, 62
- tf_jiggle(), 78
- tf_landmarks_extrema, 63
- tf_landmarks_extrema(), 43, 55, 72, 74, 77, 82
- tf_mfpc_efunctions (tfb_mfpc), 25
- tf_mfpc_efunctions(), 27
- tf_mfpc_scores (tfb_mfpc), 25
- tf_mfpc_scores(), 27
- tf_minmax, 50, 65, 69
- tf_mv_methods, 18, 27, 28, 37, 46, 58, 66
- tf_mv_unimplemented, 37
- tf_ncomp (tf_mv_methods), 66
- tf_ncomp(), 37
- tf_norm (tf_geom), 57
- tf_order, 67
- tf_order(), 50, 65
- tf_rebase, 69
- tf_rebase(), 26, 61, 66, 89
- tf_register, 70
- tf_register(), 43, 53, 55, 64, 66, 73, 74, 76, 77, 82
- tf_register_shape, 73
- tf_register_shape(), 43, 55, 64, 72, 74, 76, 77, 82
- tf_registration, 43, 55, 64, 70–72, 74, 74, 82
- tf_reparam_arclength (tf_geom), 57
- tf_rgp, 77
- tf_rgp(), 63
- tf_rotations (tf_registration), 74
- tf_rotations(), 74
- tf_scales (tf_registration), 74
- tf_scales(), 74
- tf_smooth, 78
- tf_smooth(), 66
- tf_sparsify (tf_jiggle), 62
- tf_speed (tf_geom), 57
- tf_speed(), 46
- tf_split, 80
- tf_splitcombine (tf_split), 80
- tf_tangent (tf_geom), 57
- tf_template (tf_registration), 74
- tf_template(), 70, 71, 74
- tf_warp, 81
- tf_warp(), 43, 55, 64, 66, 72, 74, 77
- tf_where, 83
- tf_zoom, 85
- tf_zoom(), 7, 14, 48, 66, 86
- tfb, 21
- tfb(), 9, 24, 27, 28, 33, 52, 62, 90
- tfb_fpc, 22
- tfb_fpc(), 8, 9, 21, 22, 26, 27, 33
- tfb_mfpc, 25
- tfb_mfpc(), 9, 18, 24, 26, 28, 37, 46, 58, 66
- tfb_mv, 27
- tfb_mv(), 18, 26, 27, 37, 46, 58, 66
- tfb_spline, 29
- tfb_spline(), 9, 21, 22, 24
- tfb_wavelet (tfb), 21
- tfbrackets ([.tf_mv), 90
- tfd, 33
- tfd(), 34, 37, 42, 44, 62, 82, 90
- tfd_mv, 36
- tfd_mv(), 18, 26–28, 46, 58, 66
- tfgroupgenerics, 38
- tfsummaries, 8, 11, 15, 40
- unique_id, 86
- unique_id(), 7, 14, 48, 85
- var (tfsummaries), 40

vctrs, [87](#)
 vctrs::vec_arith(), [38](#)
 vctrs::vec_cast(), [90](#)
 vctrs::vec_ptype2(), [90](#)
 vec_arith.tfb(tfgroupgenerics), [38](#)
 vec_arith.tfd(tfgroupgenerics), [38](#)
 vec_cast.tfb_fpc.tfb_fpc(vctrs), [87](#)
 vec_cast.tfb_fpc.tfb_spline(vctrs), [87](#)
 vec_cast.tfb_fpc.tfd_irreg(vctrs), [87](#)
 vec_cast.tfb_fpc.tfd_reg(vctrs), [87](#)
 vec_cast.tfb_mv.tfb_mv(vctrs), [87](#)
 vec_cast.tfb_mv.tfd_mv(vctrs), [87](#)
 vec_cast.tfb_spline.tfb_fpc(vctrs), [87](#)
 vec_cast.tfb_spline.tfb_spline(vctrs), [87](#)
 vec_cast.tfb_spline.tfd_irreg(vctrs), [87](#)
 vec_cast.tfb_spline.tfd_reg(vctrs), [87](#)
 vec_cast.tfd_irreg.tfb_fpc(vctrs), [87](#)
 vec_cast.tfd_irreg.tfb_spline(vctrs), [87](#)
 vec_cast.tfd_irreg.tfd_irreg(vctrs), [87](#)
 vec_cast.tfd_irreg.tfd_reg(vctrs), [87](#)
 vec_cast.tfd_mv.tfb_mv(vctrs), [87](#)
 vec_cast.tfd_mv.tfd_mv(vctrs), [87](#)
 vec_cast.tfd_reg.tfb_fpc(vctrs), [87](#)
 vec_cast.tfd_reg.tfb_spline(vctrs), [87](#)
 vec_cast.tfd_reg.tfd_irreg(vctrs), [87](#)
 vec_cast.tfd_reg.tfd_reg(vctrs), [87](#)
 vec_ptype2.tfb_fpc.tfb_fpc(vctrs), [87](#)
 vec_ptype2.tfb_fpc.tfb_spline(vctrs), [87](#)
 vec_ptype2.tfb_fpc.tfd_irreg(vctrs), [87](#)
 vec_ptype2.tfb_fpc.tfd_reg(vctrs), [87](#)
 vec_ptype2.tfb_mv.tfb_mv(vctrs), [87](#)
 vec_ptype2.tfb_mv.tfd_mv(vctrs), [87](#)
 vec_ptype2.tfb_spline.tfb_fpc(vctrs), [87](#)
 vec_ptype2.tfb_spline.tfb_spline(vctrs), [87](#)
 vec_ptype2.tfb_spline.tfd_irreg(vctrs), [87](#)
 vec_ptype2.tfb_spline.tfd_reg(vctrs), [87](#)
 vec_ptype2.tfd_irreg.tfb_fpc(vctrs), [87](#)
 vec_ptype2.tfd_irreg.tfb_spline(vctrs), [87](#)
 vec_ptype2.tfd_irreg.tfd_irreg(vctrs), [87](#)
 vec_ptype2.tfd_irreg.tfd_reg(vctrs), [87](#)
 vec_ptype2.tfd_mv.tfb_mv(vctrs), [87](#)
 vec_ptype2.tfd_mv.tfd_mv(vctrs), [87](#)
 vec_ptype2.tfd_reg.tfb_fpc(vctrs), [87](#)
 vec_ptype2.tfd_reg.tfb_spline(vctrs), [87](#)
 xtfm.tf(tf_order), [67](#)
 zoo::na.approx(), [35, 43](#)
 zoo::na.fill(), [35, 43](#)
 zoo::na.locf(), [35, 43](#)
 zoo::na.spline(), [35, 43](#)
 zoo::rollmean(), [79](#)