# Package 'tf'

May 22, 2024

**Title** S3 Classes and Methods for Tidy Functional Data

**Version** 0.3.4

**Description** Defines S3 vector data types for vectors of functional data (grid-based, spline-based or functional principal components-based) with all arithmetic and summary methods, derivation, integration and smoothing, plotting, data import and export, and data wrangling, such as re-evaluating, subsetting, sub-assigning, zooming into sub-domains, or extracting functional features like minima/maxima and their locations.
The implementation allows including such vectors in data frames for joint analysis of functional and scalar variables.

**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, methods, mgcv, mvtnorm, pracma, purrr, rlang, stats, vctrs (>= 0.2.4), zoo

**Suggests** covr, dplyr, knitr, refund, testthat (>= 3.0.0)

**Encoding** UTF-8

**RoxygenNote** 7.3.1

**Collate** 'approx.R' 'bibentries.R' 'brackets.R' 'calculus.R' 'convert-construct-utils.R' 'convert.R' 'depth.R' 'evaluate.R' 'fwise.R' 'globals.R' 'graphics.R' 'interpolate.R' 'ops.R' 'math.R' 'methods.R' 'print-format.R' 'rebase.R' 'rng.R' 'smooth.R' 'soft-impute-svd.R' 'summarize.R' 'tf-package.R' 'tfb-fpc.R' 'tfb-spline.R' 'tfb-class.R' 'tfd-class.R' 'tf-s4.R' 'tfb-fpc-utils.R' 'tfb-spline-utils.R' 'utils.R' 'vctrs-cast.R' 'vctrs-ptype2.R' 'where.R' 'zoom.R'

**Config/testthat/edition** 3

**NeedsCompilation** no

**Author** Fabian Scheipl [aut, cre] (<https://orcid.org/0000-0001-8172-3603>),
Jeff Goldsmith [aut],
Julia Wrobel [ctb] (<https://orcid.org/0000-0001-6783-1421>),
Maximilian Muecke [ctb] (<https://orcid.org/0009-0000-9432-9795>),
Sebastian Fischer [ctb] (<https://orcid.org/0000-0002-9609-3197>),
Trevor Hastie [ctb] (softImpute author),
Rahul Mazumder [ctb] (softImpute author),
Chen Meng [ctb] (mogsa author)

**Maintainer** Fabian Scheipl <fabian.scheipl@googlemail.com>

**Repository** CRAN

**Date/Publication** 2024-05-22 10:30:03 UTC

## R topics documented:

`as.data.frame.tf` *Convert functional data back to tabular data formats*

## 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

| | |
|---|---|
| `x` | a `tf` object |
| `row.names` | `NULL` or a character vector giving the row names for the data frame. Missing values are not allowed. |
| `optional` | not used |
| `unnest` | if `TRUE`, the function will return a data.frame with the evaluated functions. |
| `...` | additional arguments to be passed to or from methods. |
| `arg` | a vector of argument values / evaluation points for `x`. Defaults to `tf_arg(x)`. |
| `interpolate` | should functions be evaluated (i.e., inter-/extrapolated) for values in `arg` for which no original data is available? Only relevant for the raw data class `tfd`, for which it defaults to `FALSE`. Basis-represented functional data `tfb` 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` for the (unique) names of `x` or a numeric identifier, `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

## `ensure_list` *Turns any object into a list*

### Description

See above.

### Usage

```
ensure_list(x)
```

### Arguments

| | |
|---|---|
| `x` | any input |

### Value

`x` turned into a list.

### See Also

Other tidyfun developer tools: `prep_plotting_arg()`, `unique_id()`

## `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 eigen*functions* $\phi_j(t)$, not eigen*vectors* $\phi_j = (\phi_j(t_1), \ldots, \phi_j(t_n))$, specifically:
$\int_T \phi_j(t)^2 dt \approx \sum_i \Delta_i \phi_j(t_i)^2 = 1$ given quadrature weights $\Delta_i$, not $\phi_j' \phi_j = \sum_i \phi_j(t_i)^2 = 1$;
$\int_T \phi_j(t)\phi_k(t) dt = 0$ not $\phi_j' \phi_k = \sum_i \phi_j(t_i)\phi_k(t_i) = 0$, c.f. `mogsa::wsvd()`.
For incomplete data, this uses an adaptation of `softImpute::softImpute()`, see references. Note that 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)

Trevor Hastie, Rahul Mazumder, Cheng Meng, Fabian Scheipl

## References

code adapted from / inspired by `mogsa::wsvd()` by Cheng Meng and `softImpute::softImpute()` by Trevor Hastie and Rahul Mazumder.
Meng C (2023). `mogsa`*: Multiple omics data integrative clustering and gene set analysis*. `https://bioconductor.org/packages/mogsa`.
Mazumder, Rahul, Hastie, Trevor, Tibshirani, Robert (2010). “Spectral regularization algorithms for learning large incomplete matrices.” *The Journal of Machine Learning Research*, **11**, 2287-2322.
Hastie T, Mazumder R (2021). `softImpute`*: Matrix Completion via Iterative Soft-Thresholded SVD*. R package version 1.4-1, `https://CRAN.R-project.org/package=softImpute`.

## See Also

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

Other tfb_fpc-class: `tfb_fpc()`

## functionwise *Summarize each tf in a vector*

### Description

These functions extract (user-specified) **function-wise** summary statistics from each entry in a tf-vector. To summarize a vector of functions at each argument value, see ?tfsummaries. Note that these will tend to yield lots of NAs for irregular tfd 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)

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

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))

tf_fvar(x, arg = tf_arg(x))

tf_fsd(x, arg = tf_arg(x))

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

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

### Arguments

| | |
|---|---|
| x | a tf object |
| .f | a function or formula that is applied to each entry of x, see purrr::as_mapper() and Details. |
| arg | defaults to standard argument values of x |
| ... | additional arguments for purrr::as_mapper() |
| na.rm | a logical 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_fmin()`: minimal value of each function
- `tf_fmedian()`: median value of each 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_fvar()`: variance of each function: $\frac{1}{|T|}\int_T (x_i(t) - \bar{x}(t))^2 dt$
- `tf_fsd()`: standard deviation of each function: $\sqrt{\frac{1}{|T|}\int_T (x_i(t) - \bar{x}(t))^2 dt}$
- `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: `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, ~ quantile(.x$value, .8)) |> unlist()
# minimal value of each function for t >.5
tf_fwise(x, ~ min(.x$value[.x$arg > .5])) |> unlist()
```

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

## `in_range` *Find out if values are inside given bounds*

### 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: `tf_arg()`, `tf_zoom()`

## `Ops.tf` *Math, Summary and Ops Methods for* `tf`

### Description

These methods and operators mostly work `arg`-value-wise on `tf` objects, see `?groupGeneric` for implementation details.

**Usage**

```
## S3 method for class 'tf'
Ops(e1, e2)

## 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'
Ops(e1, e2)

## S3 method for class 'tfb'
Ops(e1, e2)

## 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 |
| `x` | an `tf` |
| `...` | `tf`-objects (not used for `Math` group generic) |

## Details

See examples below. 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 on the evaluation grid used for computation. 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.

## 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
```

| | |
|---|---|
| plot.tf | base *plots for* tf*s* |

## 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` | an `tf` object |
| `y` | (optional) numeric vector to be used as `arg` (i.e., for the **x**-axis...!) |
| `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`grDevices::rgb()`) for noodle transparency. Defaults to 2/(no. of observations). Lower is more transparent. |
| `...` | additional arguments for `matplot()` ("spaghetti") or `image()` ("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.

## `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

### See Also

Other tidyfun developer tools: `ensure_list()`, `unique_id()`

## print.tf *Pretty printing and formatting for functional data*

### Description

Print/format tf-objects.

### Usage

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

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

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

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

## S3 method for class 'tf'
format(
  x,
  digits = 2,
  nsmall = 0,
  width = options()$width,
  n = 5,
  prefix = TRUE,
  ...
)
```

### Arguments

| | |
|---|---|
| x | any R object (conceptually); typically numeric. |
| n | how many elements of x to print out |
| ... | further arguments passed to or from other methods. |
| 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 *minimum* field width or `NULL` or `0` for no restriction.<br>`AsIs` method: the *maximum* field width for non-character objects. `NULL` corresponds to the default `12`. |
| `prefix` | used internally. |

## Value

prints out `x` and returns it invisibly

---

`tfb` *Constructors for functional data in basis representation*

---

## Description

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

## Usage

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

tfb_wavelet(data, ...)

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

## Arguments

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

## 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 $B_k(t); k = 1, \ldots, K$ identical for all observations and weight or coefficient vectors $b_i = (b_{i1}, \ldots, b_{iK})$ 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()`

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

---

`tfbrackets` *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'
x[i, j, interpolate = TRUE, matrix = TRUE]

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

## Arguments

| | |
|---|---|
| `x` | an `tf` |
| `i` | index of the observations (`integer`ish, `character` or `logical`, usual R rules apply) |
| `j` | The `arg` used to evaluate the functions. A (list of) `numeric` vectors. *NOT* interpreted as a column number but as the argument value of the respective functional datum. |
| `interpolate` | should functions be evaluated (i.e., inter-/extrapolated) for values in `arg` for which no original data is available? Only relevant for the raw data class `tfd`, for which it defaults to `TRUE`. Basis-represented `tfb` are always "interpolated". |
| `matrix` | should the result be returned as a `matrix` or as a list of `data.frame`s? If `TRUE`, `j` has to be a (list of a) single vector of `arg`. See return value. |
| `value` | `tf` object for subassignment. This is typed more strictly than concatenation: subassignment only happens if the common type of `value` and `x` is the same as the type of `x`, so subassignment never changes the type of `x` but may do a potentially lossy cast of `value` to the type of `x` (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 > length(x)

All of the above will trigger errors.

## Value

If j is missing, 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
# 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))
```

---

# 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

| | |
|---|---|
| `data` | a `matrix`, `data.frame` or `list` of suitable shape, or another `tf`-object containing functional data. |
| `...` | arguments to the `method` which computes the (regularized/smoothed) FPCA - see e.g. `fpc_wsvd()`. Unless set by the user, uses proportion of variance explained `pve = 0.995` to determine the truncation levels. |
| `id` | The name or number of the column defining which data belong to which function. |
| `arg` | `numeric`, or list of `numeric`s. 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 (`1:<length of data>`) as the default if not provided. |
| `value` | The name or number of the column containing the function evaluations. |
| `domain` | range of the `arg`. |
| `method` | the function to use that computes eigenfunctions and scores. Defaults to `fpc_wsvd()`, 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()`

## 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::fpca.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::fpca_sc for use as FPCA method in tfb_fpc --
# 1. define scoring function (simple weighted LS fit)
fpca_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_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 '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

| | |
|---|---|
| `data` | a `matrix`, `data.frame` or `list` of suitable shape, or another `tf`-object containing functional data. |
| `...` | arguments to the calls to `mgcv::s()` setting up the basis (and to `mgcv::magic()` or `mgcv::gam.fit()` if `penalized = TRUE`). Uses `k = 25` cubic regression spline basis functions (`bs = "cr"`) by default, but should be set appropriately by the user. See Details and examples in the vignettes. |
| `id` | The name or number of the column defining which data belong to which function. |
| `arg` | `numeric`, or list of `numeric`s. 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 (`1:<length of data>`) as the default if not provided. |
| `value` | The name or number of the column containing the function evaluations. |
| `domain` | range of the `arg`. |
| `penalized` | `TRUE` (default) estimates regularized/penalized basis coefficients via `mgcv::magic()` or `mgcv::gam.fit()`, `FALSE` yields ordinary least squares / ML estimates for basis coefficients. `FALSE` is much faster but will overfit for noisy data if `k` is (too) large. |
| `global` | Defaults to `FALSE`. If `TRUE` and `penalized = TRUE`, 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(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()`

---

`tfd` *Constructors for vectors of "raw" functional data*

---

### Description

Various constructor methods for `tfd`-objects.

`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 function information by default: (`id`, `arg`, `value`)

`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

### 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

| | |
|---|---|
| `data` | a `matrix`, `data.frame` or `list` of suitable shape, or another `tf`-object. when this argument is `NULL` (i.e. when calling `tfd()`) this returns a prototype of class `tfd` |
| `...` | not used in `tfd`, except for `tfd.tf` – specify `arg` and `interpolate = TRUE` to turn an irregular `tfd` into a regular one, see examples. |
| `arg` | `numeric`, or list of `numeric`s. 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 (`1:<length of data>`) as the default if not provided. |
| `domain` | range of the `arg`. |
| `evaluator` | a function accepting arguments `x, arg, evaluations`. See details for `tfd()`. |
| `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

`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` and
- `tf_approx_nocb` for "next observation carried backward" (i.e., `zoo::na.locf()` with `na.rm = FALSE, fromLast = TR`
  See `tf:::zoo_wrapper` and `tf:::tf_approx_linear`, which is simply `zoo_wrapper(zoo::na.tf_approx, na.rm = FALSE)`, for examples of implementations of this.

## Value

an `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()
```

---

## `tfsummaries` *Functions that summarize* `tf` *objects across argument values*

---

### 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 = c("MBD", "pointwise"), ...)

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, ...)
```

### Arguments

| | |
|---|---|
| `x` | a `tf` object |
| `...` | optional additional arguments. |
| `na.rm` | logical. Should missing values be removed? |
| `depth` | method used to determine the most central element in `x`, i.e., the median. One of the functional data depths available via `tf_depth()` or `"pointwise"` for a pointwise median function. |
| `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 variance function, the functional median, and the functional range (i.e., *pointwise* min/max) of the central half of the functions, as defined by `tf_depth()`.

### See Also

`tf_fwise()`

Other tidyfun summary functions: `functionwise`

---

## `tf_approx_linear` *Inter- and extrapolation functions for* `tfd`*-objects*

---

### Description

These are the currently available `evaluator`-functions for `tfd`-objects, which control how the entries are inter-/extrapolated to previously unseen `arg`-values. They all are merely wrappers around `zoo::na.fill()`, `zoo::na.approx()`, etc... Note that these are not meant to be called directly – they are internal functions used by `tf_evaluate.tfd()` to do its thing.

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` and
- `tf_approx_nocb` for "next observation carried backward" (i.e., `zoo::na.locf()` with `na.rm = FALSE, fromLast = TR`

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 $(x_i, f(x_i))$=`(arg, evaluations)`-tuples at new argument values `x`.

## See Also

tfd

Other tidyfun inter/extrapolation functions: `tf_evaluate()`, `tf_interpolate()`

Other tidyfun inter/extrapolation functions: `tf_evaluate()`, `tf_interpolate()`

Other tidyfun inter/extrapolation functions: `tf_evaluate()`, `tf_interpolate()`

Other tidyfun inter/extrapolation functions: `tf_evaluate()`, `tf_interpolate()`

Other tidyfun inter/extrapolation functions: tf_evaluate(), tf_interpolate()

Other tidyfun inter/extrapolation functions: tf_evaluate(), tf_interpolate()

---

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.

### 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'
```

```
is.na(x)

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

is_tf(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)
```

## 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`.<br>**for `tf_arg<-`:** (list of) new `arg`-values.<br>**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` | as usual |
| `...` | dots |

## Value

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

## See Also

Other tidyfun utility functions: `in_range()`, `tf_zoom()`

## tf_depth *Functional Data Depth*

### Description

Data depths for functional data. Currently implemented: Modified Band-2 Depth, see reference.

### Usage

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

## S3 method for class 'matrix'
tf_depth(x, arg, depth = "MBD", na.rm = TRUE, ...)

## S3 method for class 'tf'
tf_depth(x, arg, depth = "MBD", na.rm = TRUE, ...)
```

### Arguments

| | |
|---|---|
| `x` | `tf` (or a matrix of evaluations) |
| `arg` | grid of evaluation points |
| `depth` | currently available: "MBD", i.e. modified band depth |
| `na.rm` | TRUE remove missing observations? |
| `...` | further arguments handed to the function computing the respective tf_depth. |

### Value

vector of tf_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.

## 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, order = 1, ...)

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

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

### Arguments

| | |
|---|---|
| `f` | a `tf`-object |
| `arg` | grid to use for the finite differences. Not the `arg` of the returned object for `tfd`-inputs, see Details. |
| `order` | order of differentiation. Maximal value for `tfb_spline` is 2. |
| `...` | not used |

### Details

The derivatives of `tfd` objects use centered finite differences, e.g. for first derivatives $f'((t_i + t_{i+1})/2) \approx \frac{f(t_i)+f(t_{i+1})}{t_{i+1}-t_i}$, so the **domains of differentiated** `tfd` **will shrink (slightly) at both ends**. 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 some spline bases like `"cr"` or `"tp"` which always 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 slightly different `arg` or `basis` for the derivatives, see Details)

### Methods (by class)

- `tf_derive(matrix)`: row-wise finite differences
- `tf_derive(tfd)`: derivatives by finite differencing.
- `tf_derive(tfb_spline)`: derivatives by finite differencing.
- `tf_derive(tfb_fpc)`: derivatives by finite differencing.

### See Also

Other tidyfun calculus functions: `tf_integrate()`

---

## `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]
```

---

`tf_integrate` *Integrals and anti-derivatives of functional data*

---

## 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,
  lower = tf_domain(f)[1],
  upper = tf_domain(f)[2],
  definite = TRUE,
  ...
)

## S3 method for class 'tfb'
tf_integrate(
  f,
  arg,
  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 `definite=TRUE`, this can be a vector of the same length as `f`. |
| `upper` | upper limits of the integration range (but see `definite` arg / Description). For `definite=TRUE`, 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. |

## 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()`

---

## `tf_interpolate` *Re-evaluate `tf`-objects on a new grid of argument values.*

---

## 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 `NA`s, 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, ...)
```

## 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_jiggle` *Make a `tf` (more) irregular*

### Description

Randomly create some irregular functional data from regular ones. **jiggle** it by randomly moving around its `arg`-values. Only for `tfd`. **sparsify** it by setting (100*`dropout`)% of its values to `NA`.

### Usage

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

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

### Arguments

| | |
|---|---|
| `f` | a `tfd` object |
| `amount` | how far away from original grid points can the new 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 |
| `...` | additional args for the returned `tfd` in `tf_jiggle` |
| `dropout` | how many values of `f` to drop, defaults to 50%. |

### Value

an (irregular) `tfd` object

### See Also

Other tidyfun RNG functions: `tf_rgp()`

Other tidyfun RNG functions: `tf_rgp()`

## `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

| | |
|---|---|
| `object` | a `tf` object whose representation should be changed |
| `basis_from` | the `tf` object with the desired basis, `arg`, `evaluator`, etc. |
| `arg` | optional new `arg` values, defaults to those of `basis_from` |
| `...` | 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.

---

`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 $\phi$, `order` $o$ and `nugget` effect variance $\sigma^2$ are:

- *squared exponential* covariance $Cov(x(t), x(t')) = \exp(-(t-t')^2)/\phi) + \sigma^2\delta_t(t')$.
- *Wiener* process covariance $Cov(x(t), x(t')) = \min(t', t)/\phi + \sigma^2\delta_t(t')$,
- *Matèrn* process covariance $Cov(x(t), x(t')) = \frac{2^{1-o}}{\Gamma(o)}(\frac{\sqrt{2o}|t-t'|}{\phi})^o \text{Bessel}_o(\frac{\sqrt{2o}|t-t'|}{s}) + \sigma^2\delta_t(t')$

## Usage

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

## 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). |
| `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 Matèrn 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". |

## Value

an `tfd`-vector of length `n`

## See Also

Other tidyfun RNG functions: `tf_jiggle()`

---

`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" (see `pracma::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` = $<$number of grid points$>$/20 (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` = $<$number of grid points$>$/10 (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)
library(pracma)
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, "rollmean", 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 = "orginal and\n savgol (red)")
lines(f_sg, col = "red")
```

---

## `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_anywhere(f, cond, arg)
```

### 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 *all* `arg` for which `cond` is true, the *first*, the *last* or their *range* or logical flags whether the functions fullfill the condition *any*where. 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`.

## 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)
```

---

## `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 iff `begin` or `end` are not scalar.

## See Also

Other tidyfun utility functions: `in_range()`, `tf_arg()`

## 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` *Make syntactically valid unique names*

## Description

See above.

### Usage

```
unique_id(x)
```

### Arguments

| | |
|---|---|
| `x` | any input |

### Value

`x` turned into a list.

### See Also

Other tidyfun developer tools: `ensure_list()`, `prep_plotting_arg()`

---

## `vec_cast.tfd_reg` `vctrs` *methods for* `tf` *objects*

---

### Description

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

### Usage

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## Arguments

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

## 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*

## Value

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

## See Also

`vctrs::vec_cast()`, `vctrs::vec_ptype2()`

# Index