

Package: tantale (via r-universe)

October 9, 2026

Title Mining and Analysis of Transcription Activator-Like Effectors

Version 0.99.0.9000

Maintainer Sebastien Cunnac <sebastien.cunnac@ird.fr>

Description A toolkit for the study of transcription activator-like effectors (TALEs) of *Xanthomonas*. `tell_tales()` finds TALE genes in genome assemblies and can correct the frameshifts that make them hard to annotate in error-prone long-read assemblies. TALEs are held in S3 classes that record each one as a series of parts (N-terminus, repeats with their repeat-variable diresidues (RVDs), C-terminus), with methods to validate, subset, combine and plot them. TALEs are aligned repeat by repeat, compared with R implementations of the 'DisTAL' and 'FuncTAL' distances <doi:10.3389/fpls.2015.00545>, and grouped into families by hierarchical clustering or k-medoids. Wrappers for 'AnnoTALE' <doi:10.1038/srep21077>, 'TALEcorrection' <doi:10.1186/s12864-023-09228-1>, 'TALVEZ' <doi:10.1371/journal.pone.0068464> and 'PrediTALE' <doi:10.1371/journal.pcbi.1007206> read their results into the same objects; the last two predict target sites in host genomes.

License MIT + file LICENSE

Copyright tantale authors, except the bundled third-party programs and data listed with their licences in inst/COPYRIGHTS

URL <https://scunnac.github.io/tantale/>,
<https://github.com/scunnac/tantale>

BugReports <https://github.com/scunnac/tantale/issues>

Depends R (>= 4.0.0)

SystemRequirements Java (>= 8), and conda, mamba or micromamba (which `tantale_setup()` uses to install MAFFT, HMMER and MMseqs2)

Imports magrittr, fs, dplyr, ggplot2 (>= 3.5.0), tibble, stringr, readr, glue, IRanges, Biostrings, pwalgn, GenomicRanges, BSgenome, plyranges, systemPipeR, DECIPHER, ape, ggtree,

tidytree, applot, universalmotif, gplots, cluster, ggnewscale,
 matrixStats, reticulate, cli, digest, methods, rlang, tidyr,
 stats, utils, grDevices, graphics, BiocGenerics, BiocParallel,
 GenomeInfoDb, S4Vectors, rtracklayer

Suggests knitr, DT, sessioninfo, biomatr, rmarkdown, quarto, covr,
 testthat (>= 3.0.0), withr, codetools

VignetteBuilder quarto

Config/Needs/website pkgdown, tidyverse/tidytemplate

Config/testthat/edition 3

Config/testthat/parallel false

Encoding UTF-8

OS_type unix

biocViews Software

Config/roxygen2/version 8.0.0

Roxygen list(markdown = TRUE)

LazyData true

Config/pak/sysreqs libcairo2-dev cmake libfontconfig1-dev libfreetype6-dev make default-jdk libbz2-
 dev libicu-dev libjpeg-dev liblzma-dev libpng-dev libuv1-dev libxml2-dev libssl-
 dev python3 libx11-dev xz-utils zlib1g-dev

Repository <https://ropensci-staging.r-universe.dev>

Date/Publication 2026-10-09 12:48:37 UTC

RemoteUrl <https://github.com/scunnac/tantale>

RemoteRef HEAD

RemoteSha 398c06635a747806ebb0337f9f1198a4a2221462

Contents

tantale-package	4
[.pairwise_distances	6
[.tales	7
as.matrix.pairwise_distances	8
as.matrix.tales_msa	9
as_tales	10
correct_tales	11
distances_assert_square	12
distances_restrict	13
format.tales	14
format.tales_msa	15
is_pairwise_distances	16
is_tales	17
is_tales_msa	17
pairwise_distances	18

plot.tales	19
plot.tales_msa	21
plot_target_preds	23
preditale	24
print.tales	26
print.tales_msa	27
run_annotale_assign	28
run_annotale_build	29
run_annotale_load_classes	31
run_annotale_predict	32
rvd_dna_specificity	33
summary.tales	35
summary.tales_msa	36
tale_annotations	37
tales	39
tales_align	40
tales_anchor_codes	42
tales_anomalies	43
tales_assert_complete	44
tales_assign_domain_codes	45
tales_bind	47
tales_coded_strings	48
tales_compare_distal	49
tales_compare_functal	51
tales_consensus	53
tales_consensus_match	54
tales_domain_codes	55
tales_domain_distances	55
tales_from_annotale	57
tales_from_telltales	58
tales_get_dna_seq	59
tales_get_protein_seq	60
tales_group_hclust	61
tales_group_kmedoids	62
tales_msa	64
tales_msa_params	65
tales_msa_width	66
tales_names	67
tales_namespace	68
tales_predict_targets	68
tales_requirements	69
tales_rvd_strings	70
tales_tale_distances	71
tales_to_universalmotif	72
talomes_heatmap	73
talvez	75
tantale_genome	77
tantale_setup	78

tell_tales	79
validate_pairwise_distances	85
validate_tales	86
validate_tales_msa	86

Index	88
--------------	-----------

tantale-package	<i>tantale: Transcription Activator-Like Effectors (TALEs) tools</i>
-----------------	--

Description

Tools to find the TALE genes of *Xanthomonas* in DNA sequences, compare and group them, align their repeat arrays and predict the plant promoter sites they bind. The [package website](#) has worked examples of each step.

Details

Finding TALEs

`tell_tales()` finds TALE genes with HMMER profiles of the TALE domains, much as AnnoTALE does, and is written for noisy sequences such as draft assemblies or long reads: it can correct frameshifts against reference TALEs before AnnoTALE reads out the RVDs. `run_annotale_predict()` runs AnnoTALE itself, and `correct_tales()` runs TALEcorrection. `tales_from_telltales()` and `tales_from_annotale()` load either result as a `tales` object.

Working with TALEs in R

A `tales` object holds one row per part (N-terminus, repeat or C-terminus) of each TALE, and `tales_anomalies()` reports the TALEs whose structure is not standard. A `tales_msa` adds the alignment of the arrays. `tale_distances` and `domain_distances` objects hold distances between TALEs or between their domains.

Comparing and grouping TALEs

`tales_compare_distal()` and `tales_compare_funcal()` reimplement the DisTAL and FuncTAL comparisons of QueTAL. `tales_group_kmedoids()` and `tales_group_hclust()` group TALEs from these distances, and `talomes_heatmap()` compares the groups across strains. AnnoTALE's own classes come from `run_annotale_build()`, and its published catalogue from `run_annotale_load_classes()` and `run_annotale_assign()`. The `tale_annotations` dataset gives 128 curated TALEs of ten *X. oryzae* genomes to compare against.

Aligning TALEs

`tales_align()` aligns the RVD or domain sequences of the arrays with MAFFT; `tales_consensus()` and the `plot()` method summarise the alignment.

Predicting targets

`tales_predict_targets()` runs Talvez (`talvez()`) or PrediTALe (`preditale()`) on DNA sequences such as promoters, and `plot_target_preds()` draws a predicted site with the RVDs facing their bases.

Setting up

MAFFT, HMMER, mmseqs2 and the Perl that Talvez runs on come from a conda environment the package builds for itself, so conda (or mamba, or micromamba) is needed for most of the package. `tantale_setup()` called bare reports what is present and changes nothing; `tantale_setup(install = TRUE)` builds or repairs the environment and downloads the Java tools (AnnoTALe, PrediTALe, TALEcorrection) and the example genomes. It checks tool versions as well as presence: MAFFT and HMMER are pinned because later MAFFT versions align TALE repeat strings differently.

Without conda, `reticulate::install_miniconda()` installs one from inside R. An existing conda, mamba or micromamba is found through `reticulate::conda_binary()`.

Note

tantale runs on Linux and on Intel macOS, the platforms the pinned conda tools exist for; it does not run on Windows. The Java tools need Java on the PATH, which `tantale_setup()` checks.

Author(s)

Maintainer: Sebastien Cunnac <sebastien.cunnac@ird.fr> ([ORCID](#))

Authors:

- Sebastien Cunnac <sebastien.cunnac@ird.fr> ([ORCID](#))
- Bao Tram Vi <vbt576@gmail.com> ([ORCID](#))

Other contributors:

- Alvaro L. Pérez-Quintero (Author of the bundled TALVEZ 3.2 and of the FuncTAL table behind `rvd_dna_specificity`) [copyright holder]
- Molly Megraw (Author of the PlantTFBS Java classes bundled with TALVEZ (simplescan-code/)) [copyright holder]
- Artemis G. Hatzigeorgiou (Author of the PlantTFBS Java classes bundled with TALVEZ (simplescan-code/)) [copyright holder]

See Also

Useful links:

- <https://scunnac.github.io/tantale/>
- <https://github.com/scunnac/tantale>
- Report bugs at <https://github.com/scunnac/tantale/issues>

[.pairwise_distances *Subset a pairwise_distances object*

Description

Subsets rows and columns like an ordinary tibble, but the class travels only while the result still satisfies the contract: id1/id2 (both character) and dissim (numeric) all present. Dropping any of them leaves something that can no longer be described as a pairwise distance table, and the result is a plain tibble, without an error.

Usage

```
## S3 method for class 'pairwise_distances'
x[...]
```

Arguments

x A pairwise_distances object.
 ... Passed on to the tibble/data frame method.

Details

Squareness is not checked here (see [distances_assert_square](#)): subsetting rows is an everyday way to end up with a non-square table.

Value

The subset: still a pairwise_distances (or its tale_distances/domain_distances subclass) if the contract holds, otherwise a plain tibble.

See Also

Other pairwise distances: [as.matrix.pairwise_distances\(\)](#), [distances_assert_square\(\)](#), [distances_restrict\(\)](#), [is_pairwise_distances\(\)](#), [pairwise_distances\(\)](#), [rvd_dna_specificity](#), [tales_assign_domain_codes\(\)](#), [tales_compare_distal\(\)](#), [tales_compare_functal\(\)](#), [tales_domain_distances\(\)](#), [tales_group_hclust\(\)](#), [tales_group_kmedoids\(\)](#), [tales_tale_distances\(\)](#), [validate_pairwise_distances\(\)](#)

Examples

```
d <- pairwise_distances(data.frame(
  id1 = c("A1", "A1", "A2", "A2"),
  id2 = c("A1", "A2", "A1", "A2"),
  dissim = c(0, 35, 35, 0)
))
is_pairwise_distances(d[1:2, ]) # row subsetting never breaks the contract
class(d[, "id1"])                # dropping dissim/id2: the class steps aside
```

[.tales

Subset a tales object

Description

Subsets rows and columns like an ordinary tibble, but the class travels only while the result still satisfies the `tales` contract.

Usage

```
## S3 method for class 'tales'
x[...]
```

Arguments

`x` A `tales` object.

`...` Passed on to the tibble/data frame method.

Details

Row subsetting never breaks anything: every invariant a `tales` checks still holds on any subset of rows, so filtering to one array, or to its repeats only, is still a valid `tales`. Column subsetting is where it degrades: dropping `array_id`, or both residue columns (`rvd`, `dom_code`) at once, leaves something that can no longer be described as a `tales`, and the result is a plain tibble, without an error. Dropping an optional column (`seqnames`, `aa_seq`, ...) has no such consequence. A `tales_msa` degrades one step at a time: losing `alignment_position` alone steps it back to a plain `tales`.

Attributes travel with a valid subset. `dom_code_namespace` describes the run the codes came from, so cutting an object down with `[]` keeps its namespace unchanged.

Value

The subset: still a `tales` (or `tales_msa`) if the contract holds, otherwise a plain tibble.

See Also

Other `tales` objects: [as_tales\(\)](#), [format.tales\(\)](#), [format.tales_msa\(\)](#), [is_tales\(\)](#), [new_tales\(\)](#), [print.tales\(\)](#), [print.tales_msa\(\)](#), [summary.tales\(\)](#), [summary.tales_msa\(\)](#), [tales\(\)](#), [tales_anchor_codes\(\)](#), [tales_anomalies\(\)](#), [tales_assert_complete\(\)](#), [tales_bind\(\)](#), [tales_names\(\)](#), [tales_namespace\(\)](#), [tales_requirements\(\)](#), [validate_tales\(\)](#)

Examples

```
x <- tales(data.frame(
  array_id = c("A1", "A1"), position_in_array = c(1L, 2L),
  rvd = c("NTERM", "HD")
))
```

```
is_tales(x[1, ]) # row subsetting never breaks the contract
class(x[, -1])  # dropping array_id: the class steps out of the way
```

```
as.matrix.pairwise_distances
```

Render a distance table as a square matrix

Description

Materialises the wide form, with ids as both row and column names, sorted.

Usage

```
## S3 method for class 'pairwise_distances'
as.matrix(x, value = PAIRWISE_DISTANCES_VALUE_COL, ...)
```

Arguments

x	A pairwise_distances object.
value	Name of the column to fill cells with. Defaults to dissim.
...	Ignored.

Details

x must be square (see [distances_assert_square](#), called here for you): every id present in both id1 and id2. A table filtered on one id column only is not; use [distances_restrict](#) to filter both.

Value

A numeric matrix, square, with sorted ids as dimnames.

See Also

Other pairwise distances: [\[.pairwise_distances\(\)](#), [distances_assert_square\(\)](#), [distances_restrict\(\)](#), [is_pairwise_distances\(\)](#), [pairwise_distances\(\)](#), [rvd_dna_specificity](#), [tales_assign_domain_codes\(\)](#), [tales_compare_distal\(\)](#), [tales_compare_functal\(\)](#), [tales_domain_distances\(\)](#), [tales_group_hclust\(\)](#), [tales_group_kmedoids\(\)](#), [tales_tale_distances\(\)](#), [validate_pairwise_distances\(\)](#)

Examples

```
d <- pairwise_distances(data.frame(
  id1 = c("A1", "A1", "A2", "A2"),
  id2 = c("A1", "A2", "A1", "A2"),
  dissim = c(0, 35, 35, 0)
))
as.matrix(d)
```

as.matrix.tales_msa *Render a TALE alignment as a matrix*

Description

Materialises the rectangular form: one row per array, one column per alignment position, cells holding the requested layer. This is the only place the gapped matrix is built — the long object is the canonical storage.

Usage

```
## S3 method for class 'tales_msa'
as.matrix(x, value = NULL, gap = NA, ...)
```

Arguments

x	A tales_msa object.
value	Name of the column to fill cells with. Defaults to the first available of rvd, dom_code.
gap	Value to use for gaps. Defaults to NA.
...	Ignored.

Value

A character matrix with arrays as rows and alignment positions as columns.

See Also

Other TALE alignment: [is_tales_msa\(\)](#), [tales_align\(\)](#), [tales_consensus\(\)](#), [tales_consensus_match\(\)](#), [tales_msa\(\)](#), [tales_msa_params\(\)](#), [tales_msa_width\(\)](#), [validate_tales_msa\(\)](#)

Examples

```
aligned <- data.frame(
  array_id = c("A1", "A1", "A1", "A2", "A2"),
  position_in_array = c(1L, 2L, 3L, 1L, 2L),
  alignment_position = c(1L, 2L, 3L, 1L, 3L),
  rvd = c("NTERM", "HD", "CTERM", "NTERM", "CTERM")
)
msa <- tales_msa(aligned)
as.matrix(msa) # gaps as NA
as.matrix(msa, gap = "-") # gaps as "-"
```

as_tales

*Coerce sequences of TALE parts to a tales object***Description**

Turns sep-separated TALE sequences — RVD strings, or repeat-code strings — into a [tales](#) object, taking sequence order as `position_in_array` and sequence names as `array_id`.

Usage

```
as_tales(x, ...)

## S3 method for class 'data.frame'
as_tales(x, ...)

## Default S3 method:
as_tales(x, sep = "-", residue_col = c("rvd", "dom_code"), ...)
```

Arguments

<code>x</code>	A path to a fasta file, a BStringSet/AStringSet, a list of strings, or a data frame (which is passed to tales , including a <code>tales_msa</code> , which is demoted).
<code>...</code>	Passed to methods.
<code>sep</code>	Separator between elements of a sequence. Use "-" for RVD sequences and " " for repeat-code strings.
<code>residue_col</code>	Which residue column the parsed elements become: "rvd" (default) or "dom_code". It is never guessed from the values.

Details

The result is deliberately column-poor: a bare sequence file carries no domain types, amino acid sequences or source contigs, so only `array_id`, `position_in_array` and the chosen residue column are produced. That is a valid `tales`: `seqnames` and the other columns are optional.

A `tales_msa` passed to `as_tales()` is demoted to a plain `tales`: `alignment_position` stays as an ordinary column, and the alignment width is dropped (`tales_msa_width` returns `NULL`).

Value

A validated `tales` object.

See Also

Other `tales` objects: [\[.tales\(\)](#), [format.tales\(\)](#), [format.tales_msa\(\)](#), [is_tales\(\)](#), [new_tales\(\)](#), [print.tales\(\)](#), [print.tales_msa\(\)](#), [summary.tales\(\)](#), [summary.tales_msa\(\)](#), [tales\(\)](#), [tales_anchor_codes\(\)](#), [tales_anomalies\(\)](#), [tales_assert_complete\(\)](#), [tales_bind\(\)](#), [tales_names\(\)](#), [tales_namespace\(\)](#), [tales_requirements\(\)](#), [validate_tales\(\)](#)

Examples

```
rvd_fasta <- system.file("extdata", "TalA_RVDSeqs_AnnoTALE.fasta",
                          package = "tantale")
as_tales(rvd_fasta, sep = "-")
```

correct_tales

Correct TALE ORFs in error-prone sequences

Description

A much faster alternative to running [tell_tales](#) in correction mode on error-prone sequences such as ONT-assembled genomes: a wrapper around the Java TALE correction tool from this [GitHub page](#). It takes an input fasta file and writes a file with corrected indels in TALE coding sequences using an approach described in the Erkes et al. [paper](#). [tell_tales](#) can subsequently be run on the corrected sequences in no correction mode.

Usage

```
correct_tales(
  uncorrected_path,
  corrected_path = tempfile("corrected_tales_", fileext = ".fa"),
  hmm_path = .tantale_tool("talecorrection_hmm"),
  return_corrections = FALSE,
  conda_bin = "auto",
  java_args = ""
)
```

Arguments

uncorrected_path	Path to the input sequence file
corrected_path	Path of the output file. The default is a new file under tempdir() , which R deletes when the session ends: give a path to keep the corrected sequences.
hmm_path	Path to the folder containing the profile HMM files. The default points to the ones built from <i>Xanthomonas oryzae</i> pv. <i>oryzae</i> (Xoo) templates, in the tools tantale_setup() downloads; Xoc (<i>X. oryzae</i> pv. <i>oryzicola</i>) ones are in the sibling folder Xoc. Please see the GitHub page for instructions on building custom profiles.
return_corrections	Specify TRUE if you want the list of executed operations on the sequence as a tibble.
conda_bin	Path to your Conda binary file if you need to specify a path different from the one that is automatically searched by the reticulate package functions.
java_args	A single string of options for the Java virtual machine, placed before <code>-jar</code> . The default, <code>""</code> , leaves Java's own defaults (a heap of up to a quarter of the physical memory); give a <code>-Xmx</code> value, such as <code>"-Xmx8G"</code> , if TALEcorrection runs out of memory. TALEcorrection itself takes no options beyond its inputs.

Value

A tibble if return_corrections is TRUE or the path to the corrected sequences file.

See Also

Other TALE discovery: [tale_annotations](#), [tales_from_annotale\(\)](#), [tales_from_telltales\(\)](#), [tell_tales\(\)](#)

Examples

```
# Needs nhmmer and a Java runtime.
subj <- system.file("extdata", "bai3_sample_tal_genomic_regions.fasta",
                    package = "tantale")
out_fa <- tempfile(fileext = ".fa")
correct_tales(uncorrected_path = subj, corrected_path = out_fa)
```

distances_assert_square

Assert that a distance table is complete and square

Description

Checks that the table holds every ordered pair of the ids it contains — n^2 rows for n ids. Phrased over the ids *present*, so that a symmetric subset stays square: filtering both id columns to the same set of entities preserves this, while filtering one of them does not.

Usage

```
distances_assert_square(x, arg = "x")
```

Arguments

x	A pairwise_distances object.
arg	Name of the argument being checked, for the error message.

Details

Squareness is a **precondition** of the functions that need it. The class itself does not require it, because filtering the two id columns one after the other passes through a legitimate non-square intermediate.

Value

x, invisibly.

See Also

Other pairwise distances: `[.pairwise_distances(), as.matrix.pairwise_distances(), distances_restrict(), is_pairwise_distances(), pairwise_distances(), rvd_dna_specificity, tales_assign_domain_codes(), tales_compare_distal(), tales_compare_functal(), tales_domain_distances(), tales_group_hclust(), tales_group_kmedoids(), tales_tale_distances(), validate_pairwise_distances()`

Examples

```
d <- pairwise_distances(data.frame(
  id1 = c("A1", "A1", "A2", "A2"),
  id2 = c("A1", "A2", "A1", "A2"),
  dissim = c(0, 35, 35, 0)
))
distances_assert_square(d)

try(distances_assert_square(d[1:3, ])) # missing the A2-A2 pair -- errors
```

distances_restrict	<i>Restrict a distance table to a set of entities</i>
--------------------	---

Description

Filters **both** id columns, which is what keeps the result square.

Usage

```
distances_restrict(x, ids)
```

Arguments

x	A pairwise_distances object.
ids	A character vector of entity ids to keep.

Value

A pairwise_distances over ids only.

See Also

Other pairwise distances: `[.pairwise_distances(), as.matrix.pairwise_distances(), distances_assert_square(), is_pairwise_distances(), pairwise_distances(), rvd_dna_specificity, tales_assign_domain_codes(), tales_compare_distal(), tales_compare_functal(), tales_domain_distances(), tales_group_hclust(), tales_group_kmedoids(), tales_tale_distances(), validate_pairwise_distances()`

Examples

```
d <- pairwise_distances(data.frame(
  id1 = c("A1", "A1", "A1", "A2", "A2", "A2", "A3", "A3", "A3"),
  id2 = c("A1", "A2", "A3", "A1", "A2", "A3", "A1", "A2", "A3"),
  dissim = c(0, 35, 60, 35, 0, 40, 60, 40, 0)
))
distances_restrict(d, c("A1", "A2"))
```

format.tales

Render a tales object as lines of text

Description

The representation `print.tales` emits, returned as a character vector, so an object's description can go somewhere other than the console: a log, an error message, a report.

It shows what the class knows and a tibble would not: the number of arrays as distinct from the number of parts, which residue layers the object carries, the `dom_code` namespace when it is stamped, and a preview of the first and last arrays as sequences.

The preview uses `dom_code` when present and `rvd` otherwise. Domain codes are the more discriminating of the two: two arrays can share an RVD sequence while being built from different repeats.

This replaces the `format()` a `tales` would otherwise inherit from `tibble`. Call `format(tibble::as_tibble(x))` for that one.

Usage

```
## S3 method for class 'tales'
format(x, n = 2L, ...)
```

Arguments

<code>x</code>	A <code>tales</code> object.
<code>n</code>	Number of arrays to show at each end. Everything is shown when the object holds $2 * n$ arrays or fewer.
<code>...</code>	Unused, present for compatibility with the <code>format</code> generic.

Value

A character vector, one element per line.

See Also

Other `tales` objects: `[.tales()`, `as_tales()`, `format.tales_msa()`, `is_tales()`, `new_tales()`, `print.tales()`, `print.tales_msa()`, `summary.tales()`, `summary.tales_msa()`, `tales()`, `tales_anchor_codes()`, `tales_anomalies()`, `tales_assert_complete()`, `tales_bind()`, `tales_names()`, `tales_namespace()`, `tales_requirements()`, `validate_tales()`

Examples

```
parts <- data.frame(
  array_id = c("A1", "A1", "A1", "A2", "A2", "A2"),
  position_in_array = c(1L, 2L, 3L, 1L, 2L, 3L),
  domain_type = c("N-terminus", "repeat", "C-terminus",
                  "N-terminus", "repeat", "C-terminus"),
  rvd = c("NTERM", "HD", "CTERM", "NTERM", "NI", "CTERM")
)
x <- tales(parts)
format(x)
```

format.tales_msa	<i>Render a tales_msa object as lines of text</i>
------------------	---

Description

As [format.tales](#), plus the alignment width, and a preview of aligned rows: gaps are drawn and every cell padded to a common width, so the columns line up down the page.

Usage

```
## S3 method for class 'tales_msa'
format(x, n = 2L, gap = "-", ...)
```

Arguments

x	A tales_msa object.
n	Number of arrays to show at each end.
gap	What to draw where an array has no residue at a position.
...	Unused, present for compatibility with the format generic.

Value

A character vector, one element per line.

See Also

Other tales objects: [\[.tales\(\)](#), [as_tales\(\)](#), [format.tales\(\)](#), [is_tales\(\)](#), [new_tales\(\)](#), [print.tales\(\)](#), [print.tales_msa\(\)](#), [summary.tales\(\)](#), [summary.tales_msa\(\)](#), [tales\(\)](#), [tales_anchor_codes\(\)](#), [tales_anomalies\(\)](#), [tales_assert_complete\(\)](#), [tales_bind\(\)](#), [tales_names\(\)](#), [tales_namespace\(\)](#), [tales_requirements\(\)](#), [validate_tales\(\)](#)

Examples

```
aligned <- data.frame(
  array_id = c("A1", "A1", "A1", "A2", "A2"),
  position_in_array = c(1L, 2L, 3L, 1L, 2L),
  alignment_position = c(1L, 2L, 3L, 1L, 3L),
  rvd = c("NTERM", "HD", "CTERM", "NTERM", "CTERM")
)
msa <- tales_msa(aligned)
format(msa)
```

is_pairwise_distances *Is this a pairwise distance table?*

Description

Is this a pairwise distance table?

Usage

```
is_pairwise_distances(x)
```

Arguments

x An object.

Value

A logical scalar.

See Also

Other pairwise distances: [\[.pairwise_distances\(\)](#), [as.matrix.pairwise_distances\(\)](#), [distances_assert_square\(\)](#), [distances_restrict\(\)](#), [pairwise_distances\(\)](#), [rvd_dna_specificity](#), [tales_assign_domain_codes\(\)](#), [tales_compare_distal\(\)](#), [tales_compare_functal\(\)](#), [tales_domain_distances\(\)](#), [tales_group_hclust\(\)](#), [tales_group_kmedoids\(\)](#), [tales_tale_distances\(\)](#), [validate_pairwise_distances\(\)](#)

Examples

```
d <- data.frame(
  id1 = c("A1", "A1", "A2", "A2"),
  id2 = c("A1", "A2", "A1", "A2"),
  dissim = c(0, 35, 35, 0)
)
is_pairwise_distances(d)
is_pairwise_distances(pairwise_distances(d))
```

is_tales	<i>Is this a tales object?</i>
----------	--------------------------------

Description

Is this a tales object?

Usage

```
is_tales(x)
```

Arguments

x An object.

Value

A logical scalar.

See Also

Other tales objects: [\[.tales\(\)](#), [as_tales\(\)](#), [format.tales\(\)](#), [format.tales_msa\(\)](#), [new_tales\(\)](#), [print.tales\(\)](#), [print.tales_msa\(\)](#), [summary.tales\(\)](#), [summary.tales_msa\(\)](#), [tales\(\)](#), [tales_anchor_codes\(\)](#), [tales_anomalies\(\)](#), [tales_assert_complete\(\)](#), [tales_bind\(\)](#), [tales_names\(\)](#), [tales_namespace\(\)](#), [tales_requirements\(\)](#), [validate_tales\(\)](#)

Examples

```
parts <- data.frame(
  array_id = c("A1", "A1", "A1"),
  position_in_array = 1:3,
  domain_type = c("N-terminus", "repeat", "C-terminus"),
  rvd = c("NTERM", "HD", "CTERM")
)
is_tales(parts)                      # a plain data frame
is_tales(tales(parts))
```

is_tales_msa	<i>Is this a tales_msa object?</i>
--------------	------------------------------------

Description

Is this a tales_msa object?

Usage

```
is_tales_msa(x)
```

Arguments

`x` An object.

Value

A logical scalar.

See Also

Other TALE alignment: `as.matrix.tales_msa()`, `tales_align()`, `tales_consensus()`, `tales_consensus_match()`, `tales_msa()`, `tales_msa_params()`, `tales_msa_width()`, `validate_tales_msa()`

Examples

```
aligned <- data.frame(
  array_id = c("A1", "A1", "A1", "A2", "A2"),
  position_in_array = c(1L, 2L, 3L, 1L, 2L),
  alignment_position = c(1L, 2L, 3L, 1L, 3L),
  rvd = c("NTERM", "HD", "CTERM", "NTERM", "CTERM")
)
msa <- tales_msa(aligned)
is_tales_msa(msa)
is_tales_msa(as_tales(msa)) # back to a plain tales
```

pairwise_distances	Create a pairwise distance table
--------------------	----------------------------------

Description

The long form of a square pairwise distance over one entity set: one row per ordered pair of entities, with a dissim value (0 for identical entities, larger for more different ones).

Usage

```
pairwise_distances(x, dom_code_namespace = NULL)
```

```
tales_distances(x, dom_code_namespace = NULL)
```

```
domain_distances(x, dom_code_namespace = NULL)
```

Arguments

`x` A data frame with `id1`, `id2` and `dissim` columns. The function also accepts a similarity: it converts a table carrying `sim` or `norm_arlem_score` instead of a distance into `dissim` and drops the similarity columns, so that only one copy of the quantity remains. It keeps any further column (`arlem_score`, `max_length`, or anything else).

dom_code_namespace

Optional scalar string identifying the run whose dom_code values this table is keyed by, see [tales_namespace](#). Relevant for domain_distances(), whose ids are dom_codes.

Details

tale_distances() and domain_distances() are the entity-specific flavours: distances between whole TALE arrays, and between distinct domains, repeats and the two termini alike. For domain_distances, dissim is the percentage of amino acids that differ between two domains (see [tales_domain_distances](#)); for tale_distances, it is the array alignment cost between two TALEs (see [tales_tale_distances](#)). The two flavours add no structure, only meaning: every method is written once on the parent.

Value

A validated pairwise_distances object.

See Also

Other pairwise distances: [\[.pairwise_distances\(\)\]](#), [as.matrix.pairwise_distances\(\)](#), [distances_assert_square\(\)](#), [distances_restrict\(\)](#), [is_pairwise_distances\(\)](#), [rwd_dna_specificity](#), [tales_assign_domain_codes\(\)](#), [tales_compare_distal\(\)](#), [tales_compare_functal\(\)](#), [tales_domain_distances\(\)](#), [tales_group_hclust\(\)](#), [tales_group_kmedoids\(\)](#), [tales_tale_distances\(\)](#), [validate_pairwise_distances\(\)](#)

Examples

```
d <- data.frame(
  id1 = c("A1", "A1", "A2", "A2"),
  id2 = c("A1", "A2", "A1", "A2"),
  dissim = c(0, 35, 35, 0)
)
pairwise_distances(d)
domain_distances(d, dom_code_namespace = "example")

# A similarity is folded into the distance.
similar <- data.frame(id1 = "A1", id2 = "A2", sim = 65)
tale_distances(similar)
```

plot.tales

Plot the domain composition of a set of TALE arrays

Description

A compact, information-rich view of the arrays in a tales object: one point per part, positioned by its place in the array, outlined by domain type and filled by part type and amino-acid length. Each repeat carries its RVD, each terminus the short name of its code (N-, -C, or ?? for a terminus that does not match its TALE domain profile; see [tales_anchor_codes](#)). Arrays are listed from the top in alphabetical order of array_id.

The fill colours follow the role of each part. The canonical 34-aa repeat and the 20-aa half-repeat that ends every array get calm colours, so a repeat of any other length (an aberrant repeat, for instance) stands out. N-termini are shades of wine and C-termini shades of teal, lighter when shorter, which makes a truncated terminus visible. The legend gives the length of each colour, its keys outlined like the parts they stand for. All colours are chosen to stay distinct for colour-blind readers.

A `tales_msa` dispatches to `plot.tales_msa` instead, being the more specific class.

Usage

```
## S3 method for class 'tales'
plot(x, position = c("array", "alignment"), facet_by = "seqnames", ...)
```

Arguments

<code>x</code>	A <code>tales</code> object, as returned by <code>tales_from_telltales</code> or in the <code>tales</code> element of <code>tales_compare_distal</code> 's output. A data frame of parts is accepted and converted with <code>tales</code> .
<code>position</code>	Which coordinate to lay the parts out on. "array" (default) uses <code>position_in_array</code> , so each array starts at 1 and runs contiguously. "alignment" uses <code>alignment_position</code> , which requires an aligned object (or one demoted from a <code>tales_msa</code> , which keeps the column): gaps then appear as empty columns and shared features line up. Aberrant repeats, for instance, are visible as a column in the aligned layout and scattered in the unaligned one.
<code>facet_by</code>	Names of the columns whose values split the plot into panels, stacked in rows whose height follows the number of arrays. Each column must hold one value per array: "seqnames" (the default, one panel per source sequence) or "strain" for a set of genomes, for instance, and <code>c("strain", "seqnames")</code> for both. NULL draws a single panel, as does the default when <code>x</code> has no <code>seqnames</code> column.
<code>...</code>	Unused, present for compatibility with the <code>plot</code> generic.

Value

A ggplot object. Like any ggplot, it is drawn when printed, which happens automatically at the console; inside a loop or a function, call `print()` on it.

See Also

Other TALE plots: `plot.tales_msa()`, `plot_target_preds()`, `talomes_heatmap()`

Examples

```
x <- tales_from_telltales(system.file("extdata", "tellTaleExampleOutput",
                                     package = "tantale"))

plot(x)
plot(x, facet_by = NULL)
```

plot.tales_msa	<i>Plot a multiple alignment of TALEs</i>
----------------	---

Description

Draws the alignment as a heatmap: one row per array, one column per alignment position, with cells coloured by one layer and optionally labelled with another.

Usage

```
## S3 method for class 'tales_msa'
plot(
  x,
  fill = NULL,
  label = NULL,
  tale_distances = NULL,
  domain_distances = NULL,
  h_cut = 10,
  ref_pattern = NULL,
  consensus = FALSE,
  fill_type = "domain_clust",
  ...
)
```

Arguments

x	A tales_msa object.
fill	Layer whose values colour the cells. Defaults to "dom_code" when present, otherwise the first available residue layer.
label	Layer whose values are written in the cells. Left unset it defaults to "rvd" when that is not already the fill; pass NULL explicitly for an unlabelled heatmap.
tale_distances	Pairwise distances between whole TALEs, as the tale_distances element of a tales_compare_distal result. Used to build the tree panel that orders the alignment rows.
domain_distances	Pairwise distances between distinct domains, as the domain_distances element of a tales_compare_distal result. Used to group domains into clusters, and to score each domain against the reference TALE's domain at the same alignment column.
h_cut	Height at which the domain tree is cut to define clusters. Interpreted on a distance scale, so 0 means identical.
ref_pattern	Regular expression matched against the array names to choose the reference TALE. Must identify exactly one.
consensus	Whether to add a consensus panel above the alignment. The consensus is the most frequent element in each column of the labelled layer, so it always matches what the cells say.

fill_type	One of "domain_clust", "domain_sim" or "rvd_sim". The first two colour cells by domain cluster or by protein-sequence similarity to the reference, for every distinct part, termini included (see Details). "rvd_sim" colours them instead by how alike each RVD's <i>DNA-binding preference</i> is to the reference TALE's RVD at that position, on a diverging scale over [-1, 1]. The domain- and RVD-level views genuinely differ: repeats carrying HD and ND differ in sequence yet both favour cytosine, while repeats differing only at positions 12-13 are near-identical proteins targeting different bases.
...	Unused, present for compatibility with the plot generic.

Details

A `tales_msa` carries every layer at once (`rvd`, `dom_code` and whatever else the object holds), so `fill` and `label` simply name two of them.

Three settings act independently, and it helps to read the figure that way: what each cell *says*, what colour that text is, and what colour the block behind it is.

Cell text is whatever label names, or nothing when `label = NULL`. Termini are relabelled N- and -C; an unidentified terminus keeps its XXXXX code. Domain codes are padded to three characters so columns line up.

Text colour always answers one question: does this element match the consensus of its column? Black for yes, red for no, grey where the column has no consensus (a gap, or a tie). The block fills are all pale, so the text reads on every one of them. The consensus is the most frequent element in the column (`tales_consensus`), taken over the labelled layer, so the text colour and the text itself always describe the same thing.

Block fill is what `fill_type` selects, and it is the only part that can be unavailable:

fill_type	shows	needs
"domain_clust"	which cluster the domain falls in, cut at <code>h_cut</code>	<code>domain_distances</code>
"domain_sim"	protein-sequence similarity to the reference, 0-100	<code>domain_distances</code>
"rvd_sim"	how alike the RVD's DNA-binding preference is to the reference's, -1 to 1	a label layer

Every value here scores across the whole `tales_msa`, termini included: `domain_distances` covers every distinct part sequence, and "domain_clust"/"domain_sim" colour a terminus cell exactly like a repeat cell.

With no `domain_distances` and no `label`, every block is flat grey: the text still carries the consensus comparison, but there is nothing to colour blocks by.

A cell with no value for the chosen layer keeps its text and is filled grey. In "rvd_sim" that is the termini, which have no DNA-binding preference and so no position on a specificity scale. The RVD similarity table is TALVEZ's and covers 17 RVDs; a rarer RVD (NV, say) scores 1 where it is identical to the reference's RVD and is grey elsewhere.

The reference matters for both similarity fills. `ref_pattern` must match exactly one array name; otherwise the function warns and uses the default, which is the array with the most non-gap parts, ties broken alphabetically. The reference row is marked with a trailing `_#`.

Two panels may be attached. Supplying `tales_distances` with more than one array adds a dendrogram panel on the left; `consensus = TRUE` adds a consensus panel on top. When either is present

the return value is an `aplot` composition: modify the alignment through its `plotlist` element, since layers added to the composition itself do not reach it.

Value

The alignment plot, returned invisibly after being printed as a side effect: a `ggplot` normally, or, when `tales_distances` and/or `consensus` add extra panels, an `aplot` composition – see Details.

See Also

Other TALE plots: `plot.tales()`, `plot_target_preds()`, `talomes_heatmap()`

Examples

```
aligned <- data.frame(
  array_id = c("A1", "A1", "A1", "A2", "A2"),
  position_in_array = c(1L, 2L, 3L, 1L, 2L),
  alignment_position = c(1L, 2L, 3L, 1L, 3L),
  rvd = c("NTERM", "HD", "CTERM", "NTERM", "CTERM")
)
msa <- tales_msa(aligned)
plot(msa)
```

plot_target_preds	<i>Plot TALE RVD sequences along a potential DNA target region</i>
-------------------	--

Description

This function enables visual inspection of TALE target predictions results that are located **within** a specified subject DNA sequence region

Usage

```
plot_target_preds(preds, subj_file, filter_range)
```

Arguments

<code>preds</code>	A tibble of prediction results obtained with <code>preditale</code> or <code>talvez</code> or a custom table in this format.
<code>subj_file</code>	The fasta file of subject DNA sequences that was used to predict DNA binding elements.
<code>filter_range</code>	A length one genomic ranges in the form of a properly formatted character string (eg. "chr2:56-125") or an atomic <code>GenomicRanges</code> object. This argument specifies the DNA region that will be plotted together with predicted binding TALEs RVD sequences whose predicted EBE lies entirely within .

Details

RVDs sequences predicted to target an EBE on the sense strand of the DNA sequence are plotted on top of the double stranded DNA sequence in parallel to its cognate EBE which is highlighted on the corresponding strand of the DNA sequence. Those predicted to target an EBE on the opposite strand are displayed below.

Each RVD sits in a box whose colour says how strongly the RVD is expected to bind the base at that position, compared with the other three bases. The RVDs labelled "OO" correspond to the first non-canonical repeat also called repeat zero in TALE protein sequences.

Numeric values inside the boxes located immediately to the right of the TALE labels reflect prediction scores.

Value

Returns a ggplot object that can be further altered using ggplot2 package functions.

See Also

Other target prediction: [preditale\(\)](#), [tales_predict_targets\(\)](#), [talvez\(\)](#)

Other TALE plots: [plot.tales\(\)](#), [plot.tales_msa\(\)](#), [talomes_heatmap\(\)](#)

Examples

```
# Needs a Java runtime, for preditale().
x <- tales_from_telltales(system.file("extdata", "tellTaleExampleOutput",
                                     package = "tantale"))

rvds <- tales_rvd_strings(x)
subj <- system.file("extdata", "cladeIII_sweet_promoters.fasta",
                   package = "tantale")
preds <- preditale(rvd_seqs = rvds, subj_file = subj)
best <- preds[order(-preds$score), ][1, ]
# The site with 10 bp on either side
plot_target_preds(preds = best, subj_file = subj,
                  filter_range = paste0(best$subj_seq_id, ":",
                                         best$start - 10, "-", best$end + 10))
```

preditale

Run TALE target predictions on DNA sequence(s) using PrediTale

Description

An R wrapper around the **PrediTale** 'PrediTale.jar preditale' module. Takes a list of TALE RVD sequences and a fasta file of DNA sequences and runs PrediTale.

Usage

```

preditale(
  rvd_seqs,
  subj_file,
  opt_param = "",
  output_dir = NULL,
  java_args = "-Xms512M -Xmx2G",
  predictor_path = .tantale_tool("preditale")
)

```

Arguments

<code>rvd_seqs</code>	Tale RVD sequences are supplied as either a fasta file (atomic character vector) with Tale info (name) in title and sequences of RVD as a space or '-' separated string or as a Biostrings XStringSet with sequences of RVD similarly formatted. See the PrediTale man page for how to encode RVDs present on aberrant repeats.
<code>subj_file</code>	Expects a character vector specifying the path to the fasta file holding subject DNA sequence(s).
<code>opt_param</code>	A single string of options for PrediTale, as key=value pairs separated by spaces, such as "Strand=\"forward strand\"". The default, "", keeps PrediTale's own: target sites on both strands (Strand) with a penalty of 0.01 on the reverse one (r), and a prediction threshold (t) from a significance level of 1e-4 (sl), estimated on a sub-sample of the subject sequences (b). The PrediTale page lists the alternatives (a number of expected sites, n; dedicated background sequences, bs). The keys this function sets itself (TALES, s, outdir) are refused: use <code>rvd_seqs</code> , <code>subj_file</code> and <code>output_dir</code> .
<code>output_dir</code>	Expects a character vector specifying the path to an output directory. If not supplied, output files will be temporary.
<code>java_args</code>	A single string of options for the Java virtual machine, placed before <code>-jar</code> . The default starts Java with 512 MB of memory and lets it grow to 2 GB; raise <code>-Xmx</code> if PrediTale runs out of memory on a large subject.
<code>predictor_path</code>	Path to "PrediTale.jar". The default is the copy tantale_setup() downloads; give a path to use another version.

Value

A tibble with one row per predicted EBE (effector binding element), the columns renamed from the tool's own output so that [talvez](#) and [preditale](#) return the same ones in the same order: `tale_id`, `rvds`, `subj_seq_id`, `start`, `end`, `strand`, `ebe_seq` and `score`, followed by the tool's own column: `pval`, the approximate p-value PrediTale gives each site.

See Also

Other target prediction: [plot_target_preds\(\)](#), [tales_predict_targets\(\)](#), [talvez\(\)](#)

Examples

```
# Needs a Java runtime.
x <- tales_from_telltales(system.file("extdata", "tellTaleExampleOutput",
                                     package = "tantale"))

rvds <- tales_rvd_strings(x)
subj <- system.file("extdata", "cladeIII_sweet_promoters.fasta",
                   package = "tantale")
preds <- preditalc(rvd_seqs = rvds, subj_file = subj)
head(preds)
```

print.tales	<i>Print a tales object</i>
-------------	-----------------------------

Description

Emits what [format.tales](#) builds; see there for what is shown and why.

Usage

```
## S3 method for class 'tales'
print(x, n = 2L, ...)
```

Arguments

x	A tales object.
n	Number of arrays to show at each end. Everything is shown when the object holds $2 * n$ arrays or fewer.
...	Unused, present for compatibility with the <code>format</code> generic.

Value

x, invisibly.

See Also

Other tales objects: [\[.tales\(\)](#), [as_tales\(\)](#), [format.tales\(\)](#), [format.tales_msa\(\)](#), [is_tales\(\)](#), [new_tales\(\)](#), [print.tales_msa\(\)](#), [summary.tales\(\)](#), [summary.tales_msa\(\)](#), [tales\(\)](#), [tales_anchor_codes\(\)](#), [tales_anomalies\(\)](#), [tales_assert_complete\(\)](#), [tales_bind\(\)](#), [tales_names\(\)](#), [tales_namespace\(\)](#), [tales_requirements\(\)](#), [validate_tales\(\)](#)

Examples

```
parts <- data.frame(
  array_id = c("A1", "A1", "A1", "A2", "A2", "A2"),
  position_in_array = c(1L, 2L, 3L, 1L, 2L, 3L),
  domain_type = c("N-terminus", "repeat", "C-terminus",
                  "N-terminus", "repeat", "C-terminus"),
  rvd = c("NTERM", "HD", "CTERM", "NTERM", "NI", "CTERM")
)
x <- tales(parts)
print(x)
```

print.tales_msa	<i>Print a tales_msa object</i>
-----------------	---------------------------------

Description

Emits what `format.tales_msa` builds; see there for what is shown and why.

Usage

```
## S3 method for class 'tales_msa'
print(x, n = 2L, gap = "-", ...)
```

Arguments

x	A <code>tales_msa</code> object.
n	Number of arrays to show at each end.
gap	What to draw where an array has no residue at a position.
...	Unused, present for compatibility with the <code>format.generic</code> .

Value

x, invisibly.

See Also

Other tales objects: `[.tales()`, `as_tales()`, `format.tales()`, `format.tales_msa()`, `is_tales()`, `new_tales()`, `print.tales()`, `summary.tales()`, `summary.tales_msa()`, `tales()`, `tales_anchor_codes()`, `tales_anomalies()`, `tales_assert_complete()`, `tales_bind()`, `tales_names()`, `tales_namespace()`, `tales_requirements()`, `validate_tales()`

Examples

```
aligned <- data.frame(
  array_id = c("A1", "A1", "A1", "A2", "A2"),
  position_in_array = c(1L, 2L, 3L, 1L, 2L),
  alignment_position = c(1L, 2L, 3L, 1L, 3L),
  rvd = c("NTERM", "HD", "CTERM", "NTERM", "CTERM")
)
msa <- tales_msa(aligned)
print(msa)
```

run_annotale_assign	<i>Assign TALEs to AnnoTALE's published classes</i>
---------------------	---

Description

An R wrapper around the 'AnnoTALEcli.jar assign' call. AnnoTALE places each TALE in the class of the catalogue it belongs to, and opens a new class for any that fit none. Where [run_annotale_build](#) groups the TALEs you hand it among themselves, knowing nothing of what the rest of the world calls them, this one answers "which published class is this TALE in", so its names can be compared with the literature's.

Usage

```
run_annotale_assign(
  fasta_file,
  class_builder,
  output_dir = tempfile("annotale_assign_"),
  strain = NULL,
  accession = NULL,
  opt_param = "",
  java_args = "-Xms512M -Xmx8G",
  annotale_jar = .tantale_tool("annotale")
)
```

Arguments

fasta_file	Path to a FASTA file of the TALEs to assign, one record per TALE (or per TALE part), in one of these forms: • full-length TALE coding sequences, such as the Predict/TALE_DNA_sequences_*.fasta file run_annotale_predict writes; • the corresponding protein sequences, such as its Predict/TALE_protein_sequences_*.fasta; • the parts of each TALE, one record per N-terminus, repeat and C-terminus, as in its Analyze/TALE_DNA_parts.fasta or Analyze/TALE_Protein_parts.fasta; • RVD sequences, the RVDs separated by hyphens (NI-HD-NG-NN-...), as in its Analyze/TALE_RVDs.fasta or the shipped Sample_TALEs_RVDSeqs_AnnoTALE.fasta (system.file("extdata", package = "tantale")).
------------	--

class_builder	Path to the class builder XML holding the classes to assign against, as run_annotale_load_classes returns.
output_dir	Directory where output will be written, created if it does not exist. The default is a new directory under tempdir() , which R deletes when the session ends: give a path to keep the results.
strain	The strain the TALEs come from. AnnoTALE uses it to build the systematic names it proposes, so without it they are unnamed.
accession	The genome's accession number, recorded in the report.
opt_param	A single string of further options, as key=value pairs separated by spaces. The keys this function sets itself (t, c, s, a, outdir) are refused: use fasta_file, class_builder, strain, accession and output_dir.
java_args	A single string of options for the Java virtual machine, placed before -jar. The default raises the heap to 8 GB, which reading the catalogue needs; raise -Xmx if it runs out of memory.
annotale_jar	Path to the AnnoTALE jar file. The default is the copy tantale_setup() downloads; give a path to use another version.

Value

output_dir, invisibly. It holds the per-TALE assignment report, a table of original against proposed systematic names, a report and figure for each class that changed or was created, and a class builder extended with the TALEs given.

See Also

[run_annotale_load_classes](#), which fetches the classes; [run_annotale_build](#), which builds classes from scratch instead.

Other external TALE tools: [run_annotale_build\(\)](#), [run_annotale_load_classes\(\)](#), [run_annotale_predict\(\)](#)

Examples

```
# Needs a Java runtime and the catalogue, which takes a while to fetch.
classes <- run_annotale_load_classes(output_dir = tempfile("annotale_classes_"))
predicted <- run_annotale_predict(tantale_genome("MAI1"))
tales <- list.files(file.path(predicted, "Predict"),
                  pattern = "^TALE_DNA_sequences_", full.names = TRUE)
run_annotale_assign(tales, class_builder = classes, strain = "MAI1")
```

run_annotale_build	<i>Run the "build" stage of AnnoTALE</i>
--------------------	--

Description

An R wrapper around the [AnnoTALE](#) 'AnnoTALE.jar build' program. It usually takes its input from the file generated by the [run_annotale_predict](#) function.

Usage

```
run_annotale_build(
  fasta_file,
  output_dir = tempfile("annotale_build_"),
  opt_param = "c=5 s=0.01",
  java_args = "-Xms512M -Xmx6G",
  annotale_jar = .tantale_tool("annotale")
)
```

Arguments

fasta_file	Path to a TALE_DNA_sequences_*.fasta file, as written by run_annotale_predict 's "predict" stage, to be classified into groups.
output_dir	Directory where output will be written, created if it does not exist. The default is a new directory under tempdir() , which R deletes when the session ends: give a path to keep the results.
opt_param	A single string of options for AnnoTALE build, as key=value pairs separated by spaces. The default writes out the tool's own defaults: c, the cutoff on the distance that defines the maximum extent of a TALE class (5), and s, the significance level on the alignment p-value (0.01). An option left out of a string of your own keeps its default. The keys this function sets itself (t, outdir) are refused: use fasta_file and output_dir.
java_args	A single string of options for the Java virtual machine, placed before -jar. The default starts Java with 512 MB of memory and lets it grow to 6 GB; raise -Xmx if build runs out of memory.
annotale_jar	Path to the AnnoTALE jar file. The default is the copy tantale_setup() downloads; give a path to use another version.

Value

output_dir, invisibly. If AnnoTALE exits with a non-zero status, the function stops with an error of class tantale_error_annotale_failed.

See Also

Other external TALE tools: [run_annotale_assign\(\)](#), [run_annotale_load_classes\(\)](#), [run_annotale_predict\(\)](#)

Examples

```
# Needs a Java runtime and the tools tantale_setup() downloads. Chained
# from run_annotale_predict()'s own output, since that is the file build
# normally classifies.
fasta <- tantale_genome("MAI1")
predict_out <- file.path(tempdir(), "annotale_build_example", "predict")
run_annotale_predict(fasta, output_dir = predict_out)
predicted <- list.files(file.path(predict_out, "Predict"),
  pattern = "^TALE_DNA_sequences_", full.names = TRUE)
build_out <- file.path(tempdir(), "annotale_build_example", "build")
```

```
run_annotale_build(predicted, output_dir = build_out)
list.files(build_out)
```

run_annotale_load_classes

Download AnnoTALE's catalogue of TALE classes

Description

A TALE class groups the TALEs of different strains that are similar enough to be considered the same effector, and carries the systematic name the literature uses for it (TalAA, TalAB, ...). AnnoTALE curates that catalogue and publishes it; this is an R wrapper around its 'AnnoTALEcli.jar loadAndView' call, which downloads the current definition. [run_annotale_assign](#) consumes what it writes.

Usage

```
run_annotale_load_classes(
  output_dir = tempfile("annotale_classes_"),
  class_builder = NULL,
  opt_param = "",
  java_args = "-Xms512M -Xmx8G",
  annotale_jar = .tantale_tool("annotale")
)
```

Arguments

output_dir	Directory where the catalogue will be written, created if it does not exist. The default is a new directory under tempdir() , which R deletes when the session ends: given how long the download takes, give a path you keep.
class_builder	Path to a class builder XML to read instead of downloading, as written by an earlier call. NULL, the default, downloads the current definition.
opt_param	A single string of further options for the tool, as key=value pairs separated by spaces. The keys this function sets itself (c, cb, outdir) are refused: use class_builder and output_dir.
java_args	A single string of options for the Java virtual machine, placed before -jar. The default raises the heap to 8 GB, which the catalogue needs (rebuilding it was seen to hold 1.8 GB resident); raise -Xmx if it runs out of memory.
annotale_jar	Path to the AnnoTALE jar file. The default is the copy tantale_setup() downloads; give a path to use another version.

Details

The download is slow and bulky: about a quarter of an hour, and some 440 MB of output, most of it the class builder itself, which carries a cached alignment between every pair of catalogued TALEs. The file is therefore kept once and reused rather than fetched per call, which is what `output_dir` is for: point it at a directory you keep, and pass the returned path to `run_annotale_assign` as often as you like.

Record the date along with any class you take from it. A class name is stable, but the catalogue grows, and the index AnnoTALE appends to the class when it names a TALE (TalAH30, the thirtieth member of TalAH) is only true of the catalogue that produced it.

Value

The path of the class builder XML, invisibly, so the call can be passed straight to `run_annotale_assign`. Beside it, `output_dir` also holds `Lists_of_classes`, `_strains_and_TALEs/`, whose `List_of_classes.txt` gives every class as plain text, one line per member with its aligned repeat-variable diresidues, strain and systematic name. That file answers "which class is this TALE in" without going through `run_annotale_assign` at all, when the TALE is already catalogued.

See Also

`run_annotale_assign`, which places TALEs into these classes.

Other external TALE tools: `run_annotale_assign()`, `run_annotale_build()`, `run_annotale_predict()`

Examples

```
# Slow: downloads and rebuilds the catalogue, about 15 minutes.
classes <- run_annotale_load_classes(output_dir = tempfile("annotale_classes_"))
```

`run_annotale_predict` *Runs the "predict" and "analyze" steps of AnnoTALE on a fasta file*

Description

An R wrapper around the **AnnoTALE** 'AnnoTALE.jar predict' and 'AnnoTALE.jar analyze' shell calls. `run_annotale_build` completes the AnnoTALE workflow.

Usage

```
run_annotale_predict(
  fasta_file,
  output_dir = tempfile("annotale_predict_"),
  prefix = NULL,
  opt_param = "Sensitive=false",
  java_args = "",
  annotale_jar = .tantale_tool("annotale")
)
```

Arguments

fasta_file	Path to a fasta file of the DNA sequences (e.g. a genome assembly) to search for TALEs.
output_dir	Directory where output will be written, created if it does not exist. The default is a new directory under <code>tempdir()</code> , which R deletes when the session ends: give a path to keep the results.
prefix	A scalar character vector containing a prefix that will be appended to TALE names by AnnoTALE. If not supplied, the function will try to guess the prefix from the input file name.
opt_param	A single string of options for the predict stage, as key=value pairs separated by spaces. AnnoTALE predict has one: Sensitive, false by default; "Sensitive=true" runs its sensitive scan. The keys this function sets itself (g, s, outdir) are refused: use fasta_file, prefix and output_dir. The analyze stage has no option of its own beyond a run name.
java_args	A single string of options for the Java virtual machine, placed before -jar in both stages. The default, "", leaves Java's own defaults (a heap of up to a quarter of the physical memory); give a -Xmx value, such as "-Xmx8G", if AnnoTALE runs out of memory.
annotale_jar	Path to the AnnoTALE jar file. The default is the copy <code>tantale_setup()</code> downloads; give a path to use another version.

Value

output_dir, invisibly, so that the call can be passed straight to `tales_from_annotale()`. If either stage exits with a non-zero status, the function stops with an error of class `tantale_error_annotale_failed`.

See Also

Other external TALE tools: `run_annotale_assign()`, `run_annotale_build()`, `run_annotale_load_classes()`

Examples

```
# Needs a Java runtime, and the tools and genomes tantale_setup() downloads.
fasta <- tantale_genome("MAI1")
out <- file.path(tempdir(), "annotale_predict_example")
run_annotale_predict(fasta, output_dir = out)
list.files(file.path(out, "Predict"))
```

rvd_dna_specificity	<i>RVD-to-DNA-binding-specificity weights</i>
---------------------	---

Description

One row per RVD, giving its relative binding weight for each of the four bases. This is the table `tales_compare_functal` stacks, one row per repeat, to build an array's position weight matrix.

Usage

```
rvd_dna_specificity
```

Format

A tibble with 404 rows and 5 columns:

rvd The two-letter RVD code. "N*" and "H*" are RVDs whose residue 13 is missing; "00" is position 0, the base just before the first repeat's target, where TALEs prefer a T; "XX" is the flat fallback row used for an unrecognised RVD.

A, C, G, T Relative binding weight for that base. The weights do not sum to one: [tales_compare_functal](#) hands them to [create_motif](#) as raw counts (type = "PCM"), which normalises internally.

Details

Taken verbatim from QueTAL FuncTAL's own table (Info/2014mat18 in FuncTAL 1.1, by A. L. Pérez-Quintero, redistributed with permission): the values are unchanged, only a header and column names added. Conceptually related to but distinct from the internal `rvdSimDf` used by [tales_align](#)'s RVD scoring: that one is a *derived* RVD-vs-RVD similarity (a correlation between two RVDs' base-preference profiles, itself built from TALVEZ's much smaller 17-RVD `mat1`), used to score repeat *substitutions* during sequence alignment. This one is the *raw*, per-RVD base preference itself, one level upstream, used to build a whole array's binding-specificity model for comparison against another array's. The two tables serve different consumers and cannot replace each other.

References

Pérez-Quintero A.L. et al. (2015). QueTAL: a suite of tools to classify and compare TAL effectors functionally and phylogenetically. *Frontiers in Plant Science* **6**, 545. doi:10.3389/fpls.2015.00545

See Also

[tales_compare_functal](#), its only consumer.

Other pairwise distances: [\[.pairwise_distances\(\)](#), [as.matrix.pairwise_distances\(\)](#), [distances_assert_square\(\)](#), [distances_restrict\(\)](#), [is_pairwise_distances\(\)](#), [pairwise_distances\(\)](#), [tales_assign_domain_codes\(\)](#), [tales_compare_distal\(\)](#), [tales_compare_functal\(\)](#), [tales_domain_distances\(\)](#), [tales_group_hclust\(\)](#), [tales_group_kmedoids\(\)](#), [tales_tale_distances\(\)](#), [validate_pairwise_distances\(\)](#)

Examples

```
rvd_dna_specificity[rvd_dna_specificity$rvd %in% c("HD", "NI", "NG", "NN"), ]
```

summary.tales	<i>Summarise a tales object</i>
---------------	---------------------------------

Description

The measures worth knowing about a set of TALE arrays before doing anything with them, and the ones too expensive for `print.tales`.

Usage

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

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

Arguments

object	A <code>tales</code> object.
...	Unused.
x	A <code>summary.tales</code> object.

Details

Distinct domains counts different `dom_codes` against the number of parts, broken down by domain type. These are *domains*, repeats and termini alike: a `dom_code` identifies any distinct part sequence, and the two termini are parts like the repeats are. In the example output shipped with the package (see `tales_from_telltales`), 47 distinct codes cover 39 repeats and 8 termini, so reading the total as a count of distinct repeats would overstate repeat diversity.

The ratio to parts is a property of the biology: TALEs reuse repeats heavily, within an array and between arrays. It is also what decides the cost of `tales_compare_distal`, whose pairwise comparison runs over distinct domains.

Repeats per array counts repeats only, excluding the termini, so it is the number that determines how long a target box each TALE recognises. Note that `tell_tales()`'s log reports array length as the number of domain hits, which includes the two termini and is therefore larger by up to two.

Complete arrays are those where both an N- and a C-terminus were identified. An incomplete one is not necessarily wrong (the array may sit at the end of a contig, or a terminus may simply not have been detected), but completeness is a precondition several downstream functions depend on.

Anomalies come from `tales_anomalies`. They are worth computing but too slow to compute every time an object is printed, which is why they appear here.

Value

An object of class `summary.tales`: a list of the measures, so they can be used as well as read.

See Also

Other tales objects: [\[.tales\(\)\]](#), [as_tales\(\)](#), [format.tales\(\)](#), [format.tales_msa\(\)](#), [is_tales\(\)](#), [new_tales\(\)](#), [print.tales\(\)](#), [print.tales_msa\(\)](#), [summary.tales_msa\(\)](#), [tales\(\)](#), [tales_anchor_codes\(\)](#), [tales_anomalies\(\)](#), [tales_assert_complete\(\)](#), [tales_bind\(\)](#), [tales_names\(\)](#), [tales_namespace\(\)](#), [tales_requirements\(\)](#), [validate_tales\(\)](#)

Examples

```
parts <- data.frame(
  array_id = c("A1", "A1", "A1", "A2", "A2", "A2"),
  position_in_array = c(1L, 2L, 3L, 1L, 2L, 3L),
  domain_type = c("N-terminus", "repeat", "C-terminus",
                  "N-terminus", "repeat", "C-terminus"),
  rvd = c("NTERM", "HD", "CTERM", "NTERM", "NI", "CTERM")
)
x <- tales(parts)
summary(x)
```

summary.tales_msa	<i>Summarise a tales_msa object</i>
-------------------	-------------------------------------

Description

What an alignment looks like, in numbers: how gappy it is, and how much the arrays actually agree.

Usage

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

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

Arguments

object	A tales_msa object.
...	Unused.
x	A summary.tales_msa object.

Details

Columns with no consensus is the measure worth having. A column has no consensus when no element is strictly more common than the rest (see [tales_consensus](#)), so the count says how much of the alignment the arrays agree on.

It is reported per layer, and the two usually differ in a way that means something. Repeats that are distinct proteins can carry the same RVD, so a column can lack a dom_code consensus while having a clear rvd one: the arrays disagree about the repeat and agree about the base it binds. The reverse is rarer and more interesting.

Value

An object of class `summary.tales_msa`.

See Also

Other tales objects: `[.tales()`, `as_tales()`, `format.tales()`, `format.tales_msa()`, `is_tales()`, `new_tales()`, `print.tales()`, `print.tales_msa()`, `summary.tales()`, `tales()`, `tales_anchor_codes()`, `tales_anomalies()`, `tales_assert_complete()`, `tales_bind()`, `tales_names()`, `tales_namespace()`, `tales_requirements()`, `validate_tales()`

Examples

```
aligned <- data.frame(
  array_id = c("A1", "A1", "A1", "A2", "A2"),
  position_in_array = c(1L, 2L, 3L, 1L, 2L),
  alignment_position = c(1L, 2L, 3L, 1L, 3L),
  rvd = c("NTERM", "HD", "CTERM", "NTERM", "CTERM")
)
msa <- tales_msa(aligned)
summary(msa)
```

tale_annotations	<i>Curated TALE annotations for ten published Xanthomonas oryzae genomes</i>
------------------	--

Description

The TALEs of ten complete, published genomes, named as the literature names them and annotated by hand. It is the reference this package's own discovery can be checked against: three of the ten genomes, MAI1, BAI3 and PXO86, are the ones `tantale_genome()` installs, so a `tell_tales()` run on them can be compared array by array with what is recorded here.

Usage

```
tale_annotations
```

Format

A tibble with 128 rows and 10 columns. One row is one TALE gene.

strain The strain, e.g. "PXO99A". Ten of them.

label, tal_name The TALE's name in the literature, capitalised ("Tal4a") and as written in the source publication ("tal4a"). `strain` and `label` together identify a row: a `label` repeats across strains, never within one. `tal_name` is missing where the source gives no name.

annotale_class The AnnoTALE class the TALE belongs to ("TalAH"), from the catalogue of 2026-10-04. A class groups TALEs similar enough to be considered the same TALE across strains, and is how the literature refers to them. Missing for two rows, see Details.

replicon_id, genome_id The RefSeq replicon and the GenBank assembly the TALE comes from, e.g. "NC_010717" and "GCA_000019585.2". Present for every row: this is what lets a reader fetch the sequence.

pubmed PubMed identifier(s) of the publication(s) reporting the genome or the TALE, comma-separated where there is more than one.

truncTALE TRUE for the nine truncTALEs, TALEs whose C-terminal region is naturally shortened so that they no longer activate transcription. See the truncTALE article for what this does to discovery and to frameshift correction.

rvd_seq The repeat-variable diresidues in array order, dash-separated ("NI-HD-NG-..."), the format of [tales_rvd_strings](#). [tales_predict_targets](#) takes them as a named BStringSet: `Biostrings::BStringSet(setNames(rvd_seq, paste(strain, label)))`. A *lowercase* RVD ("ng", "n*") marks a repeat of non-standard length, the usual convention; 16 of the 128 arrays carry at least one, so do not upper-case this column before comparing it with anything.

unusual_feature A note, for the nine arrays that carry one, on what makes the gene atypical: repeats of non-standard length, deletions, duplications, a premature stop.

Details

The table is sparse where the sources are: tal_name is absent for 19 rows and pubmed for 19. The gaps are the state of the curation, not placeholders to be filled by a guess.

annotale_class is missing for one row: PXO99A's Tal7b, the five-repeat allele its own unusual_feature describes, too short to be catalogued.

The class is stored without the member index AnnoTALE appends to it ("TalAH", not "TalAH30"). That index numbers the members of a class as of the day it is read, and the catalogue grows: every one of the 73 indices recorded when this table was curated in 2021 had moved by 2026, while the class itself agreed in every case.

Identifiers that name an array within a single run are deliberately not included: the [tell_tales](#) array id and AnnoTALE's tempTALE name both depend on the run that produced them, so neither would survive a rerun. Matching a row to a fresh [tell_tales](#) result means matching on what the TALE is, its rvd_seq, rather than on what some run called it.

No column of DisTAL groups is included. The working file carried one, but how its values had been computed was not recorded, so they could not be vouched for. [tales_compare_distal](#) followed by [tales_group_hclust](#) computes such groups from rvd_seq.

Source

Compiled by Bao Tram Vi during the work that became this package, from the genome records and publications cited in each row; published here for the first time. annotale_class was taken from AnnoTALE's class catalogue as downloaded on 2026-10-04. The genome sequences themselves are the GenBank assemblies named in genome_id, at <https://www.ncbi.nlm.nih.gov/datasets/genome/>.

See Also

[talomes_heatmap](#), which takes a table of this shape directly: the column names below are the ones its group_col, strain_col, rvd_col and trunc_tales_col arguments expect.

Other TALE discovery: [correct_tales\(\)](#), [tales_from_annotale\(\)](#), [tales_from_telltales\(\)](#), [tell_tales\(\)](#)

Examples

```
# the TALEs of one strain
subset(tale_annotatons, strain == "PX086",
       select = c("label", "annotale_class", "truncTALE"))

# the ten strains compared class by class; truncTALEs are marked "T"
grouped <- tale_annotatons[!is.na(tale_annotatons$annotale_class), ]
talomes_heatmap(grouped, group_col = "annotale_class",
                 strain_col = "strain", rvd_col = "rvd_seq",
                 trunc_tales_col = "truncTALE")

# the naturally truncated ones, across all ten genomes
table(tale_annotatons$strain, tale_annotatons$truncTALE)

# what makes the atypical arrays atypical
subset(tale_annotatons, !is.na(unusual_feature),
       select = c("strain", "label", "unusual_feature"))
```

tales	<i>Create a tales object</i>
-------	------------------------------

Description

Builds a tales object from a data frame of TALE array parts: one row per (array, slot).

Usage

```
tales(x, dom_code_namespace = NULL, sanitize = FALSE)
```

Arguments

- | | |
|--------------------|--|
| x | A data frame with at least <code>array_id</code> and <code>position_in_array</code> columns, plus at least one of <code>rvd</code> or <code>dom_code</code> . Other recognised columns (<code>domain_type</code> , <code>position_in_crd</code> , <code>aa_seq</code> , <code>dna_seq</code> , <code>seqnames</code> , <code>source_directory</code>) are validated if present. Any further column is preserved untouched. |
| dom_code_namespace | Optional scalar string, see tales_namespace . |
| sanitize | Which arrays to remove, among those tales_anomalies lists: <ul style="list-style-type: none"> • FALSE (default): none. <code>tales()</code> warns about the anomalies, and the odd predictions stay in the object for inspection. • TRUE: the arrays whose data are inconsistent or incomplete (anomalies of kind "integrity"), with a warning naming them and why. Non-canonical TALEs, such as truncTALEs or TALEs lacking a terminus, are kept, with a message. |

- "canonical": every array with an anomaly, so that only canonical TALEs remain.

Structural corruption is an error either way.

Value

A validated tales object.

See Also

Other tales objects: `[.tales(), as_tales(), format.tales(), format.tales_msa(), is_tales(), new_tales(), print.tales(), print.tales_msa(), summary.tales(), summary.tales_msa(), tales_anchor_codes(), tales_anomalies(), tales_assert_complete(), tales_bind(), tales_names(), tales_namespace(), tales_requirements(), validate_tales()]`

Examples

```
parts <- data.frame(
  array_id = c("A1", "A1", "A1", "A2", "A2", "A2"),
  position_in_array = c(1L, 2L, 3L, 1L, 2L, 3L),
  domain_type = c("N-terminus", "repeat", "C-terminus",
                  "N-terminus", "repeat", "C-terminus"),
  rvd = c("NTERM", "HD", "CTERM", "NTERM", "NI", "CTERM")
)
tales(parts)
```

tales_align

Align the repeat arrays of a tales object

Description

Aligns TALE arrays on one of their residue layers with MAFFT's text mode, returning the alignment as a `tales_msa` — the input object with an `alignment_position` column added, so every other layer (`rvd`, `dom_code`, `aa_seq`, ...) remains available.

Usage

```
tales_align(
  x,
  residue_col = c("rvd", "dom_code"),
  domain_distances = NULL,
  mafft_opts = "--localpair --maxiterate 1000 --reorder --op 0 --ep 5 --thread 1",
  mafft_path = NULL,
  mafft_verbose = FALSE
)
```

Arguments

x	A tales object holding complete arrays.
residue_col	Which layer to align on: "rvd" (default) or "dom_code".
domain_distances	Scoring matrix for the residues being aligned. NULL (default) means none. Pass "rvd" to opt in to the built-in RVD similarity matrix when aligning RVDs, or, when aligning domain codes, a domain_distances -class object (as returned by tales_domain_distances) or the path to a *_Repeatmatrix.mat file written by Distal.
mafft_opts	Command-line options handed to MAFFT. The default, "--localpair --maxiterate 1000 --reorder --op 0 --ep 5 --thread 1", asks for an accurate local alignment and sets the gap penalties. --op is the cost of opening a gap and --ep the cost of extending one. They are set unusually here – free to open, expensive to extend – because of what is being aligned. A TALE gains or loses whole repeats, so a gap should start wherever it needs to; what should be discouraged is one long gap swallowing a stretch of repeats that really do correspond. Raise --op if the alignment fragments into too many small gaps.
mafft_path	Where to find MAFFT. NULL, the default, uses the tantale conda environment, created on first use. Give the root of a standalone MAFFT directory instead (one holding mafft.bat with the helpers under mafftdir/libexec) to use your own copy – but note that the version matters: MAFFT changed how it aligns text-mode sequences after 7.4x, and later releases leave the termini of a TALE alignment unanchored.
mafft_verbose	Whether to let MAFFT write to the console. It reports its banner, the strategy it chose and its progress through the sequences, which is dozens of lines per alignment and rarely what you want. With FALSE, that output is captured and replayed if the alignment fails, so nothing is lost for diagnosis.

Details

The mapping back from MAFFT's output is **positional**: the k-th non-gap cell of an aligned row is the k-th part fed in. That is well defined only because this function builds MAFFT's input from x itself, which is why [tales_assert_complete](#) is enforced first.

Value

A [tales_msa](#) object. The settings it was aligned with (residue_col, domain_distances, mafft_opts) are kept with it; [tales_msa_params](#) returns them.

See Also

Other TALE alignment: [as.matrix.tales_msa\(\)](#), [is_tales_msa\(\)](#), [tales_consensus\(\)](#), [tales_consensus_match\(\)](#), [tales_msa\(\)](#), [tales_msa_params\(\)](#), [tales_msa_width\(\)](#), [validate_tales_msa\(\)](#)

Examples

```
# Needs MAFFT, resolved from the tantale conda environment on first use.
rvd_fasta <- system.file("extdata", "TalA_RVDSeqs_AnnoTALE.fasta",
                        package = "tantale")
x <- as_tales(rvd_fasta, sep = "-")
msa <- tales_align(x, residue_col = "rvd")
as.matrix(msa)
```

tales_anchor_codes	<i>Codes marking a TALE array terminus</i>
--------------------	--

Description

The values a rvd column takes on non-repeat parts.

Usage

```
tales_anchor_codes()
```

Details

AnnoTALE reports as N-terminus whatever the ORF encodes upstream of the first repeat, and as C-terminus whatever it encodes downstream of the last one. [tell_tales](#) searches each of these segments with the TALE N- or C-terminal protein profile (hmmsearch). "NTERM" and "CTERM" mark a canonical TALE terminal domain, one that can be expected to do its usual job: the match has an E-value of at most terminus_max_evalue, covers at least terminus_min_cover of the profile (0.9 by default) and reaches the end of the profile that adjoins the repeats. "XXXXX" marks any other segment. An array for which AnnoTALE reported no segment on one side has no terminus part at the protein level on that side.

A terminus can fail to be canonical in several ways, which `array_report.tsv` tells apart (its `*_aa_*` and `*_dna_*` columns, see [tell_tales](#)): the segment may be unrelated sequence where the ORF starts or ends inside a frameshifted region, its repeat-side part may be in another reading frame, or the terminus may have lost a large part of its domain. The truncTALEs of *Xanthomonas oryzae* have lost the activation domain, and their C-termini are coded "XXXXX". A smaller internal deletion, such as the one in the N-terminus of TalC, a major TALE of African *X. oryzae* pv. *oryzae*, leaves a terminus canonical.

These share the rvd column with real RVDs, so code that distinguishes repeats from termini by value should use this function rather than spelling the codes out.

Value

A named character vector of the three codes.

See Also

Other tales objects: `[.tales(), as_tales(), format.tales(), format.tales_msa(), is_tales(), new_tales(), print.tales(), print.tales_msa(), summary.tales(), summary.tales_msa(), tales(), tales_anomalies(), tales_assert_complete(), tales_bind(), tales_names(), tales_namespace(), tales_requirements(), validate_tales()]`

Examples

```
tales_anchor_codes()
```

tales_anomalies	<i>Report the biological anomalies in a tales object</i>
-----------------	--

Description

Lists the arrays whose content is biologically odd:

- missing sequence data,
- a structure other than that of a standard TALE (an N-terminus, one or more repeats and a C-terminus, both termini canonical TALE terminal domains),
- impossible domain-type arrangements,
- coordinate disagreements,
- an amino acid sequence paired with more than one RVD,
- an attribute that varies within an array when it should not.

Structurally broken input (a duplicated key, a missing required column) is an error in `tales` instead.

"Odd" arrays are accepted by `tales` – real TALE predictions are messy, and refusing to load them would force cleaning outside the package and destroy the diagnostic signal. Construction warns about them and this function tells you which and why.

Each anomaly is of one of two kinds. "noncanonical" marks a TALE whose structure departs from the standard one: a terminus that is not a canonical TALE terminal domain (`terminus_noncanonical`), such as the C-terminus of a truncTALE, or no terminus part on one side (`terminus_absent`). Such TALEs can be real and interesting. "integrity" marks every other anomaly: data that are inconsistent or incomplete, or an array without repeats. `tales(x, sanitize = TRUE)` removes the arrays with an anomaly of kind "integrity" and keeps the others; `tales(x, sanitize = "canonical")` keeps canonical TALEs only.

The structure checks need a `domain_type` column, and the terminus profile check an `rvd` column; they are skipped when it is absent. The structure checks report `terminus_absent` (no N- or no C-terminus part), `no_repeat` (no repeat part) and `terminus_noncanonical` (a terminus coded XXXXX, see `tales_anchor_codes`). They apply to whole arrays: a subset keeping only the repeats is reported as lacking its termini.

Usage

```
tales_anomalies(x)
```

Arguments

`x` A `tales` object.

Value

A tibble of `array_id`, `check`, `kind` ("integrity" or "noncanonical") and `detail`, one row per anomaly. Zero rows if the object is clean.

See Also

Other `tales` objects: `[.tales()`, `as_tales()`, `format.tales()`, `format.tales_msa()`, `is_tales()`, `new_tales()`, `print.tales()`, `print.tales_msa()`, `summary.tales()`, `summary.tales_msa()`, `tales()`, `tales_anchor_codes()`, `tales_assert_complete()`, `tales_bind()`, `tales_names()`, `tales_namespace()`, `tales_requirements()`, `validate_tales()`

Examples

```
# A2 carries two N-termini -- an impossible arrangement.
odd <- data.frame(
  array_id = c("A1", "A1", "A2", "A2", "A2"),
  position_in_array = c(1L, 2L, 1L, 2L, 3L),
  domain_type = c("N-terminus", "repeat",
                  "N-terminus", "N-terminus", "repeat"),
  rvd = c("NTERM", "HD", "NTERM", "NTERM", "NI")
)
x <- suppressWarnings(tales(odd))
tales_anomalies(x)
tales(odd, sanitize = TRUE) # drops A2 instead of merely warning

# B2 is a truncTale-like array: its C-terminus is not canonical.
parts <- data.frame(
  array_id = rep(c("B1", "B2"), each = 3),
  position_in_array = rep(1:3, 2),
  domain_type = rep(c("N-terminus", "repeat", "C-terminus"), 2),
  rvd = c("NTERM", "HD", "CTERM", "NTERM", "NI", "XXXXX")
)
y <- tales(parts, sanitize = TRUE) # keeps B2, with a message
tales_anomalies(y)
z <- tales(parts, sanitize = "canonical") # drops B2
unique(z$array_id)
```

`tales_assert_complete` *Assert that a tales object holds complete arrays*

Description

Checks that every array carries all of its parts: `position_in_array` running contiguously from 1, and — when `domain_type` and `position_in_crd` are present — `position_in_crd` equal to `position_in_array` minus the number of non-repeat parts before it.

Usage

```
tales_assert_complete(x, arg = "x")
```

Arguments

x	A tales object.
arg	Name of the argument being checked, for the error message.

Details

Completeness is a **precondition** of some functions; the class itself does not require it: `filter(x, domain_type == "repeat")` legitimately produces a valid tales that is no longer complete. Alignment requires completeness, because the mapping back from a MAFFT alignment is positional — the k-th aligned residue is the k-th part fed in.

Value

x, invisibly.

See Also

Other tales objects: [\[.tales\(\)](#), [as_tales\(\)](#), [format.tales\(\)](#), [format.tales_msa\(\)](#), [is_tales\(\)](#), [new_tales\(\)](#), [print.tales\(\)](#), [print.tales_msa\(\)](#), [summary.tales\(\)](#), [summary.tales_msa\(\)](#), [tales\(\)](#), [tales_anchor_codes\(\)](#), [tales_anomalies\(\)](#), [tales_bind\(\)](#), [tales_names\(\)](#), [tales_namespace\(\)](#), [tales_requirements\(\)](#), [validate_tales\(\)](#)

Examples

```
rvd_fasta <- system.file("extdata", "TalA_RVDSeqs_AnnoTALE.fasta",
                        package = "tantale")
x <- as_tales(rvd_fasta, sep = "-")
tales_assert_complete(x) # every array is 1..n already -- no error

# Keeping only the repeats breaks completeness, which tales_align() needs.
repeats_only <- x[x$position_in_array > 1, ]
try(tales_assert_complete(repeats_only))
```

tales_assign_domain_codes

Assign a domain code to every distinct part sequence

Description

Step 1 of [tales_compare_distal\(\)](#). Gives each distinct aa_seq in x an integer code, recorded in a dom_code column, so that the parts can be compared once each rather than once per occurrence.

A code names a distinct **domain** sequence, repeats and termini alike: the N- and C-termini are parts like the repeats are, and they get codes too.

Usage

```
tales_assign_domain_codes(x)
```

Arguments

x A [tales](#) object carrying an `aa_seq` column.

Details

TALes reuse domains heavily, within an array and between arrays, so the number of distinct sequences is far smaller than the number of parts. That ratio is what makes [tales_domain_distances\(\)](#) affordable: it compares each distinct domain once.

Value

`x` with a `dom_code` column and a `dom_code_namespace` stamp.

The codes are only meaningful within one call

Codes are assigned with [dplyr::cur_group_id\(\)](#) over the distinct `aa_seq` values **present in** `x`. Add an array, remove one, or reorder the sequences, and the same protein can get a different number. They are positions in this table's own vocabulary.

So **comparing codes between two calls is an error**, and a distance table keyed by one call's codes must never be used with another call's. This is enforced: every object minted here is stamped with a `dom_code_namespace` derived from the sequences that produced it, and the classes refuse to join across namespaces. Read the stamp with [tales_namespace\(\)](#).

See Also

[tales_compare_distal\(\)](#), which runs all three steps; [tales_domain_codes\(\)](#), which reads the code-to-sequence table back out of an object that already has codes.

Other pairwise distances: [\[.pairwise_distances\(\)](#), [as.matrix.pairwise_distances\(\)](#), [distances_assert_square\(\)](#), [distances_restrict\(\)](#), [is_pairwise_distances\(\)](#), [pairwise_distances\(\)](#), [rvd_dna_specificity](#), [tales_compare_distal\(\)](#), [tales_compare_functal\(\)](#), [tales_domain_distances\(\)](#), [tales_group_hclust\(\)](#), [tales_group_kmedoids\(\)](#), [tales_tale_distances\(\)](#), [validate_pairwise_distances\(\)](#)

Examples

```
x <- tales_from_telltales(system.file("extdata", "tellTaleExampleOutput",
                                     package = "tantale"))
xa <- tales_assign_domain_codes(x)
tales_namespace(xa)
```

tales_bind	<i>Combine tales objects</i>
------------	------------------------------

Description

Row-binds one or more [tales](#) objects into one, reconciling the invariants that a plain [bind_rows](#) would not check: array_id uniqueness across inputs, the dom_code namespace, and the group column. A single input is accepted: it goes through the same reconciliation and re-validation, harmlessly.

Usage

```
tales_bind(..., on_namespace_mismatch = c("recode", "error"), sanitize = FALSE)
```

Arguments

...	One or more tales objects (not tales_msa).
on_namespace_mismatch	What to do when the inputs' dom_code namespaces (see tales_namespace) disagree. "recode" (default) drops the stale dom_code column and calls tales_assign_domain_codes on the bound result, over the union of aa_seq; "error" aborts instead, for callers who want that stricter behaviour.
sanitize	Passed to tales for the final construction: FALSE, TRUE or "canonical".

Details

There is no c() method: base c() can silently drop attributes when a tales is mixed with another object, and its signature has no room for on_namespace_mismatch.

tales_msa inputs are refused: alignment_width and alignment_position are a coordinate system specific to one alignment run, and binding two runs' matrices would produce an object that misdescribes what a given column means in each row. Demote explicitly with [as_tales](#) first if that is really what you want.

group is dropped from the result whenever present on any input: it is a clustering result over one specific distance matrix and one specific set of arrays, so two "group 1"s from separate [tales_group_hclust](#)/[tales_group_k](#) calls are not comparable. Recompute it with either on the bound result, after re-running [tales_compare_distal](#).

tale_distances/domain_distances are untouched: this function only binds tales data. A companion distance table from either input does not describe the bound object; re-run [tales_compare_distal](#) if you need one.

Value

A validated [tales](#) object.

See Also

Other tales objects: [\[.tales\(\)\]](#), [as_tales\(\)](#), [format.tales\(\)](#), [format.tales_msa\(\)](#), [is_tales\(\)](#), [new_tales\(\)](#), [print.tales\(\)](#), [print.tales_msa\(\)](#), [summary.tales\(\)](#), [summary.tales_msa\(\)](#), [tales\(\)](#), [tales_anchor_codes\(\)](#), [tales_anomalies\(\)](#), [tales_assert_complete\(\)](#), [tales_names\(\)](#), [tales_namespace\(\)](#), [tales_requirements\(\)](#), [validate_tales\(\)](#)

Examples

```
a <- tales(data.frame(
  array_id = "A1", position_in_array = 1:2,
  rvd = c("NTERM", "HD")
))
b <- tales(data.frame(
  array_id = "A2", position_in_array = 1:2,
  rvd = c("NTERM", "NI")
))
tales_bind(a, b)
```

tales_coded_strings	<i>Domain-coded strings, one per TALE array</i>
---------------------	---

Description

Renders each array as a separated string of its dom_codes in part order, where each distinct domain sequence is one "residue". [tales_align\(\)](#) and [tales_tale_distances\(\)](#) work on arrays in this form.

Usage

```
tales_coded_strings(x, sep = " ", repeats_only = FALSE)
```

Arguments

x	A tales object carrying a dom_code column.
sep	Separator between codes. Defaults to " "; see Details.
repeats_only	Drop the terminus parts, keeping only repeats. FALSE by default; see Details. Needs a domain_type column.

Details

This is the sibling of [tales_rvd_strings\(\)](#) and takes the same two arguments, but **both defaults differ**, because the two projections feed different consumers:

	tales_rvd_strings()	tales_coded_strings()
sep	"-", the AnnoTALE convention	" ", what MAFFT -text splits on
filter	repeats_only = TRUE	repeats_only = FALSE

The separator is free to choose here in a way it is not for RVDs: a `dom_code` is a bare integer rendered as text, so "1 2 3" and "1-2-3" are equally unambiguous. It defaults to a space because that is what MAFFT's text mode expects.

The termini are kept by default, where `tales_rvd_strings()` drops them. Target prediction concerns the repeat domain only, so dropping them there is right; alignment is the consumer here, and the two termini are the most reliable anchors an alignment of TALE arrays has. Set `repeats_only = TRUE` to compare bare repeat arrays.

Value

A `Biostrings::BStringSet`, named by `array_id`.

See Also

`tales_domain_codes()` for the code-to-sequence lookup, `tales_rvd_strings()` for the sibling projection.

Other tales projections: `tales_domain_codes()`, `tales_get_dna_seq()`, `tales_get_protein_seq()`, `tales_rvd_strings()`, `tales_to_universalmotif()`

Examples

```
x <- tales_from_telltales(system.file("extdata", "tellTaleExampleOutput",
                                     package = "tantale"))
xa <- tales_assign_domain_codes(x)
tales_coded_strings(xa)[1]
tales_coded_strings(xa, sep = "-", repeats_only = TRUE)[1]
```

tales_compare_distal	<i>Compute TALE and domain relatedness by domain-sequence alignment (DisTAL)</i>
----------------------	--

Description

Quantifies how TALE arrays, and the individual domains they are built from – repeats and the two termini alike – relate to one another by aligning their domain sequences. An R re-implementation of the original DisTAL Perl program, including the array alignment step (the ARLEM model, see [tales_tale_distances](#)), with parallel domain alignment, which makes it much faster (the exact speedup depends on `aln_method`).

Usage

```
tales_compare_distal(
  x,
  ncores = 1,
  aln_method = "DECIPHER",
  conda_bin = "auto"
)
```

Arguments

x	A tales object whose parts carry amino acid sequences.
ncores	Number of cores for the pairwise alignment step.
aln_method	Approach for pairwise distances between part amino acid sequences: "DECIPHER" (default), "Biostrings" or "mmseq2". See tales_domain_distances .
conda_bin	Path to a Conda binary, if reticulate cannot find it.

Details

The name says which comparison this is: [tales_compare_functal](#) compares TALEs by their predicted DNA-binding specificity instead.

It returns the two expensive products (the pairwise protein alignment between distinct domains, and the alignment of the coded arrays) with the coded input; anything else is cheap to derive from these.

This is where dom_code is minted, over the whole set of parts supplied, and where tales/domain_distances are stamped with a namespace identifying that set — see [tales_namespace](#). Passing a subset later is safe; re-running on a different part set mints different codes, and the differing namespace is what stops the two being joined by mistake. tale_distances carries no such stamp: it is keyed by array_id, a name that keeps its meaning across runs.

Value

A list of three objects, all describing the same run:

- tales: the input, with a dom_code column added.
- domain_distances: a [domain_distances](#) between distinct domains, keyed by dom_code.
- tale_distances: a [tale_distances](#) between whole arrays, keyed by array_id.

References

Pérez-Quintero A.L. et al. (2015). QueTAL: a suite of tools to classify and compare TAL effectors functionally and phylogenetically. *Frontiers in Plant Science* **6**, 545. doi:10.3389/fpls.2015.00545

Abouelhoda M.I., Giegerich R., Behzadi B., Steyaert J.-M. (2009). Alignment of minisatellite maps based on run-length encoding scheme. *Journal of Bioinformatics and Computational Biology* **7**(2), 287–308. doi:10.1142/S0219720009004060

See Also

[tales_group_hclust/tales_group_kmedoids](#) to cluster arrays from the returned tale_distances; [tales_compare_functal](#), comparing TALEs by predicted DNA-binding specificity instead of domain-sequence relatedness.

Other pairwise distances: [\[.pairwise_distances\(\), as.matrix.pairwise_distances\(\), distances_assert_square\(\), distances_restrict\(\), is_pairwise_distances\(\), pairwise_distances\(\), rvd_dna_specificity, tales_assign_domain_codes\(\), tales_compare_functal\(\), tales_domain_distances\(\), tales_group_hclust\(\), tales_group_kmedoids\(\), tales_tale_distances\(\), validate_pairwise_distances\(\)\]](#)

Examples

```
x <- tales_from_telltales(system.file("extdata", "tellTaleExampleOutput",
                                     package = "tantale"))

cmp <- tales_compare_distal(x)
names(cmp)
cmp$tales_distances
```

tales_compare_funcnal *Compare TALEs by predicted DNA-binding specificity (FuncTAL)*

Description

Quantifies how TALE arrays relate by the DNA sequence their repeats are predicted to bind. ([tales_compare_distal](#) compares them by domain sequence instead.) The function turns each array's repeats into a position weight matrix (PWM) of the bases its RVDs prefer, and compares the PWMs pairwise with [compare_motifs](#).

Usage

```
tales_compare_funcnal(
  x,
  method = "PCC",
  tryRC = FALSE,
  min.overlap = 1,
  normalise.scores = TRUE,
  score.strat = "a.mean",
  nthreads = 1
)
```

Arguments

x	A tales object carrying an rvd column.
method	One of compare_motifs() 's comparison metrics ("PCC" default). "PCC", "WPCC", "SW", "ALLR", "ALLR_LL" and "BHAT" are similarities (higher is more similar); the rest are already distances. Either kind is turned into a proper dissim in the result.
tryRC	Also score each pair's reverse-complement alignment and keep the better one. FALSE by default: an RVD array's specificity code has a fixed reading direction (5' to 3' target, N- to C-terminal repeat order), so comparing against a reverse complement asks a different, narrower biological question – do these two TALEs target opposite strands of overlapping sites. Ask it deliberately, when it is the question.
min.overlap	Minimum aligned width to accept, as compare_motifs() defines it. Defaults to 1 (any overlap at all); compare_motifs() 's own default is 6, suited to generic transcription-factor motifs, but a TALE array can have as few as half a dozen repeats, and that default would silently refuse to compare some real, short arrays.

<code>normalise.scores</code>	Penalise alignments that leave much of either motif unaligned. TRUE by default: TALE array lengths vary widely (see <code>min.overlap</code>), so an unnormalised score would favour matches that are mostly one array hanging off the end of a much longer one.
<code>score.strat</code>	How <code>compare_motifs()</code> combines per-column scores into one alignment score. "a.mean" (its own default): a sum would scale with array length, favouring long-vs-long comparisons for no biological reason.
<code>nthreads</code>	Passed to <code>compare_motifs()</code> .

Details

This is a reimplementation of QueTAL's FuncTAL comparison on `universalmotif`; the original Perl tool depends on BioPerl's `Bio::Perl` module, which current BioPerl no longer provides. **Results differ from the original FuncTAL tool, by design.** FuncTAL scored two RVD arrays by flattening their entire padded, overlapping alignment (positions and bases together) into one vector and taking a single Pearson correlation. `compare_motifs()` instead correlates matched columns individually and combines the column scores (`score.strat`); the two disagree on real data.

`tales_to_universalmotif` builds the PWMs (see its page for what drives them and how it treats an RVD missing from `rvd_dna_specificity`). It is exported so that its output can go to any `universalmotif` function.

Only a handful of `compare_motifs`'s many options are exposed here, chosen for what actually varies across TALE arrays (their differing repeat counts); see `?compare_motifs` for the rest.

Value

A `tale_distances` object, interchangeable with `tales_compare_distal`'s, so it can be handed directly to `tales_group_hclust/tales_group_kmedoids`. No tree is built here, matching `tales_compare_distal()`'s own division of labour: compare here, cluster/tree there.

See Also

`tales_compare_distal()`, comparing by domain sequence instead; `tales_to_universalmotif()`, the conversion step this uses; called directly, it lets any other `universalmotif` function (`motif_tree()`, `view_motifs()`, `scan_sequences()`, `merge_motifs()`, ...) can be run on the same PWMs.

Other pairwise distances: `[.pairwise_distances(), as.matrix.pairwise_distances(), distances_assert_square(), distances_restrict(), is_pairwise_distances(), pairwise_distances(), rvd_dna_specificity, tales_assign_domain_codes(), tales_compare_distal(), tales_domain_distances(), tales_group_hclust(), tales_group_kmedoids(), tales_tale_distances(), validate_pairwise_distances()`

Examples

```
x <- tales_from_telltales(system.file("extdata", "tellTaleExampleOutput",
                                     package = "tantale"))

fn <- tales_compare_functal(x)
fn
```

tales_consensus	<i>Compute a consensus from a TALE msa</i>
-----------------	--

Description

Pick the most frequent element in each column of the alignment matrix.

Usage

```
tales_consensus(algn)
```

Arguments

<code>algn</code>	A TALE alignment as a character matrix, one row per array and one column per alignment position, with NA for gaps. <code>as.matrix()</code> on a tales_msa returns one (see as.matrix.tales_msa), filled with RVDs or domain codes.
-------------------	--

Details

A column has a consensus only when one element is strictly more common than every other. Where two or more are tied for most frequent – as happens whenever each array carries a different repeat at that position – the result is NA, because there is no agreement to report. NA is likewise returned when the most common thing at a position is a gap.

Value

A vector of consensus elements in each column of `algn`.

See Also

Other TALE alignment: [as.matrix.tales_msa\(\)](#), [is_tales_msa\(\)](#), [tales_align\(\)](#), [tales_consensus_match\(\)](#), [tales_msa\(\)](#), [tales_msa_params\(\)](#), [tales_msa_width\(\)](#), [validate_tales_msa\(\)](#)

Examples

```
# column 1: HD is a clear majority. column 2: a three-way tie, no consensus.
aln <- matrix(c("HD", "HD", "NI",
               "NG", "NI", "HD"),
             nrow = 3, dimnames = list(c("A1", "A2", "A3"), NULL))

aln
tales_consensus(aln)

# From a tales_msa, such as tales_align() returns
msa <- tales_msa(data.frame(
  array_id = c("A1", "A1", "A1", "A2", "A2", "A3", "A3", "A3"),
  position_in_array = c(1L, 2L, 3L, 1L, 2L, 1L, 2L, 3L),
  alignment_position = c(1L, 2L, 3L, 1L, 3L, 1L, 2L, 3L),
  rvd = c("NTERM", "HD", "CTERM", "NTERM", "CTERM", "NTERM", "HD", "CTERM")
))
tales_consensus(as.matrix(msa))
```

tales_consensus_match *Do elements in a TALE msa match the consensus?*

Description

Compute a logical matrix corresponding to the input align input with TRUE where an element matches the consensus at that position and FALSE where it does not. Columns with no consensus – see [tales_consensus](#) – are NA throughout, since there is nothing there to match.

Usage

```
tales_consensus_match(align, long = TRUE)
```

Arguments

align	A TALE alignment as a character matrix, one row per array and one column per alignment position, with NA for gaps. <code>as.matrix()</code> on a tales_msa returns one (see as.matrix.tales_msa), filled with RVDs or domain codes.
long	Set to TRUE (default) to return a long tibble, or FALSE to return a logical matrix with the same shape as align.

Value

A multiple Tal sequences alignment in the form of a matrix filled with logical values if long is FALSE and a long tibble representing the original alignment otherwise (default).

See Also

Other TALE alignment: [as.matrix.tales_msa\(\)](#), [is_tales_msa\(\)](#), [tales_align\(\)](#), [tales_consensus\(\)](#), [tales_msa\(\)](#), [tales_msa_params\(\)](#), [tales_msa_width\(\)](#), [validate_tales_msa\(\)](#)

Examples

```
aln <- matrix(c("HD", "HD", "NI",
               "NG", "NI", "HD"),
             nrow = 3, dimnames = list(c("A1", "A2", "A3"), NULL))
tales_consensus_match(aln, long = FALSE)
tales_consensus_match(aln)
```

tales_domain_codes	<i>The domain code lookup table</i>
--------------------	-------------------------------------

Description

One row per distinct domain sequence: its dom_code, the amino acid sequence it stands for, and the RVD carried by that domain.

Usage

```
tales_domain_codes(x)
```

Arguments

x A [tales](#) object carrying a dom_code column.

Value

A [tibble](#) with dom_code and aa_seq columns, plus rvd when x carries one, one row per code.

See Also

[tales_coded_strings](#)

Other tales projections: [tales_coded_strings\(\)](#), [tales_get_dna_seq\(\)](#), [tales_get_protein_seq\(\)](#), [tales_rvd_strings\(\)](#), [tales_to_universalmotif\(\)](#)

Examples

```
x <- tales_from_telltales(system.file("extdata", "tellTaleExampleOutput",
                                     package = "tantale"))
xa <- tales_assign_domain_codes(x)
head(tales_domain_codes(xa))
```

tales_domain_distances

Pairwise distances between distinct TALE domains

Description

Step 2 of [tales_compare_distal\(\)](#). Aligns every distinct domain sequence against every other and returns their pairwise dissimilarity.

Usage

```
tales_domain_distances(
  x,
  aln_method = "DECIPHER",
  ncores = 1,
  conda_bin = "auto"
)
```

Arguments

<code>x</code>	A tales object carrying <code>dom_code</code> , as returned by tales_assign_domain_codes() .
<code>aln_method</code>	One of "DECIPHER" (the default), "Biostrings" or "mmseq2". "mmseq2" runs in the conda environment.
<code>ncores</code>	Number of cores for the pairwise alignment.
<code>conda_bin</code>	Passed to <code>reticulate</code> , for <code>aln_method = "mmseq2"</code> .

Details

This is the expensive step, and the one worth having on its own: the domain-level distances answer questions about domain diversity without any array-level alignment.

Distances are between **distinct domains**, keyed by `dom_code`, so the cost goes with the number of distinct sequences rather than the number of parts. See [summary\(\)](#) for that ratio on a given object.

`dissim` follows DisTAL's definition: the percentage of amino acids that change between two domains, normalised by the longer one, with residues one domain lacks counted as changes. A 20-residue half-repeat that is an exact prefix of a 34-residue repeat is therefore 14/34, about 41 percent, from it. All three backends follow this definition. They still differ slightly pair by pair: "DECIPHER" measures distances inside one multiple alignment of all domains, the other two align each pair separately, and "Biostrings" also charges a cost for opening an internal gap.

Value

A [domain_distances](#) object, stamped with `x`'s namespace.

See Also

[tales_tale_distances\(\)](#), which consumes this.

Other pairwise distances: [\[.pairwise_distances\(\)](#), [as.matrix.pairwise_distances\(\)](#), [distances_assert_square\(\)](#), [distances_restrict\(\)](#), [is_pairwise_distances\(\)](#), [pairwise_distances\(\)](#), [rvd_dna_specificity](#), [tales_assign_domain_codes\(\)](#), [tales_compare_distal\(\)](#), [tales_compare_functal\(\)](#), [tales_group_hclust\(\)](#), [tales_group_kmedoids\(\)](#), [tales_tale_distances\(\)](#), [validate_pairwise_distances\(\)](#)

Examples

```
x <- tales_from_telltales(system.file("extdata", "tellTaleExampleOutput",
                                     package = "tantale"))
xa <- tales_assign_domain_codes(x)
tales_domain_distances(xa)
```

tales_from_annotale	<i>Build a tales object from AnnoTALE's own TALE predictions</i>
---------------------	--

Description

Reads the output of AnnoTALE's analyze stage, as written by [run_annotale_predict](#), and returns a validated [tales](#) object with the same columns as [tales_from_telltales](#).

Usage

```
tales_from_annotale(
  annotale_dir,
  terminus_max_evalue = 1e-05,
  terminus_min_cover = 0.9,
  sanitize = FALSE,
  hmm_dir = system.file("extdata", "hmmProfile", package = "tantale", mustWork = TRUE)
)
```

Arguments

annotale_dir	A directory holding AnnoTALE analyze's TALE_Protein_parts.fasta, TALE_DNA_parts.fasta and TALE_RVDs.fasta, in it or in a subdirectory: the output_dir of run_annotale_predict() will do.
terminus_max_evalue, terminus_min_cover	Maximum hmmsearch E-value, and minimum fraction of the profile covered, for a terminal segment to be coded NTERM/CTERM. As in tell_tales , the match must also reach the end of the profile that adjoins the repeats.
sanitize	FALSE (default) keeps every array and warns about the anomalies; TRUE removes the arrays whose data are inconsistent or incomplete and keeps non-canonical TALEs (truncTALEs, for instance); "canonical" keeps canonical TALEs only. See tales and tales_anomalies .
hmm_dir	Directory holding the TALE terminus protein profiles.

Details

AnnoTALE predict finds TALE genes in a genome; analyze splits each one into its N-terminal region, repeats and C-terminal region, and reads the RVD of every repeat. [tell_tales](#) runs analyze too, on ORFs it delimits itself from nhmmer hits of the TALE DNA profiles, so the two need not report the same set of TALEs.

As in [tell_tales\(\)](#), each terminal segment is searched with the TALE N- or C-terminal protein profile (hmmsearch, from the tantale environment; see [tantale_setup](#)). The rvd column holds NTERM/CTERM for a segment that is a canonical terminus by that search and XXXXX for any other (see [tales_anchor_codes](#)).

array_id is AnnoTALE's name for the TALE (MAI1-tempTALE1), without the location AnnoTALE appends to it. seqnames is filled when predict's GFF3 file is found under annotale_dir.

Value

A validated tales object.

See Also

Other TALE discovery: [correct_tales\(\)](#), [tale_annotations](#), [tales_from_telltales\(\)](#), [tell_tales\(\)](#)

Examples

```
# Needs the tantale environment for hmmsearch (see tantale_setup()).
tales_from_annotale(system.file("extdata", "annotaleExampleOutput",
                                package = "tantale"))
```

tales_from_telltales	<i>Build a tales object from a tell_tales run directory</i>
----------------------	---

Description

Reads the AnnoTALE/telltale part files of a single [tell_tales](#) output directory and returns a validated [tales](#) object.

Usage

```
tales_from_telltales(telltale_dir, sanitize = FALSE)
```

Arguments

telltale_dir	Path to a single tell_tales output directory.
sanitize	FALSE (default) keeps every array and warns about the anomalies; TRUE removes the arrays whose data are inconsistent or incomplete and keeps non-canonical TALEs (truncTALEs, for instance); "canonical" keeps canonical TALEs only. See tales and tales_anomalies .

Details

One row per part: the N-terminus, each repeat and the C-terminus, as AnnoTALE split the array's longest ORF. The rvd column holds the RVD of a repeat, or a terminus code (see [tales_anchor_codes](#)): NTERM/CTERM when the terminus is a canonical TALE terminal domain by the protein profile search of [tell_tales](#), XXXXX when it is not. When AnnoTALE reported no terminus on one side, the array has no part there, with a warning. An array whose protein and DNA parts disagree is left out, with a warning.

A candidate array with a TALE terminus DNA hit that AnnoTALE could not split into parts is absent from the result, with a warning naming it. [tales_anomalies](#) cannot report such an array, since it is not in the object. After frameshift correction this usually means the array was corrected against a distant reference; see max_comparisons in [tell_tales](#).

The directory must have been written by the current version of `tell_tales()`, whose `array_report.tsv` holds the terminus check; an older one is an error.

The result carries no `dom_code`: that surrogate key is minted later, by the relatedness computation, over the whole set of parts being analysed.

Value

A validated `tales` object.

See Also

Other TALE discovery: `correct_tales()`, `tale_annotations`, `tales_from_annotale()`, `tell_tales()`

Examples

```
tales_from_telltales(system.file("extdata", "tellTaleExampleOutput",
                                package = "tantale"))
```

<code>tales_get_dna_seq</code>	<i>Whole-array DNA sequence, one per TALE array</i>
--------------------------------	---

Description

The DNA sibling of `tales_get_protein_seq()`: reassembles each array's parts, in order, into one full-length nucleotide sequence, by pasting `dna_seq` together directly. See `tales_get_protein_seq()`'s Details for why `position_in_array` is the ordering key even for a `tales_msa`.

Usage

```
tales_get_dna_seq(x)
```

Arguments

`x` A `tales` object carrying a `dna_seq` column.

Value

A `Biostrings::DNASTringSet`, named by `array_id`.

See Also

`tales_get_protein_seq()` for the protein sequence sibling.

Other `tales` projections: `tales_coded_strings()`, `tales_domain_codes()`, `tales_get_protein_seq()`, `tales_rvd_strings()`, `tales_to_universalmotif()`

Examples

```
x <- tales_from_telltales(system.file("extdata", "tellTaleExampleOutput",
                                      package = "tantale"))
tales_get_dna_seq(x)[1]
```

tales_get_protein_seq *Whole-array protein sequence, one per TALE array*

Description

Reassembles each array's parts (N-terminus, repeats and C-terminus, in order) into one full-length amino acid sequence, by pasting aa_seq together with no separator: the result is a real protein sequence, where `tales_rvd_strings()` joins RVD tokens with hyphens.

Usage

```
tales_get_protein_seq(x)
```

Arguments

x A `tales` object carrying an aa_seq column.

Details

Parts are ordered by position_in_array. An alignment never reorders an array's parts, only inserts gaps between them, and a tales_msa's gaps are never rows: a gap is a column with no row for that array. So position_in_array gives the same part order alignment_position would, without needing to skip anything, and works unchanged whether x is a bare tales or a tales_msa.

Value

An `Biostrings::AAStringSet`, named by array_id.

See Also

`tales_get_dna_seq()` for the DNA sequence sibling.

Other tales projections: `tales_coded_strings()`, `tales_domain_codes()`, `tales_get_dna_seq()`, `tales_rvd_strings()`, `tales_to_universalmotif()`

Examples

```
x <- tales_from_telltales(system.file("extdata", "tellTaleExampleOutput",
                                     package = "tantale"))
tales_get_protein_seq(x)[1]
```

tales_group_hclust	<i>Group TALEs by hierarchical clustering of their pairwise distance</i>
--------------------	--

Description

Clusters TALE arrays with `hclust` on their pairwise distance, cuts the tree into exactly `k` groups, and returns the `tales` object the distances were computed from with the result attached as its group column.

Usage

```
tales_group_hclust(x, tale_distances, k = NULL, plot_tree = FALSE)
```

Arguments

<code>x</code>	A <code>tales</code> object – the one whose comparison produced <code>tale_distances</code> .
<code>tale_distances</code>	A <code>tale_distances</code> object, as returned by <code>tales_compare_distal()</code> . A plain data frame with the same columns is also accepted and coerced.
<code>k</code>	Integer, the number of groups to cut the tree into.
<code>plot_tree</code>	Logical, whether to draw the <code>ggtree</code> dendrogram: colored by group, with a dashed line at the cut height. <code>FALSE</code> by default.

Details

The function clusters `tale_distances` but returns the `tales` object those distances came from, with the groups added. `group` is a recognised `tales` column, validated as constant within an array – it is an array-level property, like `seqnames`.

Taking `x` rather than returning a bare lookup table is what makes the correspondence checkable: the array names in `tale_distances` must be the array names in `x`, and this is the only place that can be verified. A mismatch is an error rather than a silent NA group, because a partly-grouped object is the kind of thing that fails much later and confusingly.

The bare mapping is still one line away if you want it: `unique(out[c("array_id", "group")])`.

The tree is built directly on the distances (`stats::hclust(stats::as.dist(distMat))`), matching the original DisTAL clustering this package reimplements. The tree is cut with `stats::cutree(tree, k = k)`, which always succeeds, including when a tie in merge heights would make a height-based cut ambiguous.

Value

`x` with an added (or replaced) group column.

See Also

`tales_group_kmedoids()`, the alternative method; `tales_compare_distal()`, which produces both inputs.

Other pairwise distances: `[.pairwise_distances(), as.matrix.pairwise_distances(), distances_assert_square(), distances_restrict(), is_pairwise_distances(), pairwise_distances(), rvd_dna_specificity, tales_assign_domain_codes(), tales_compare_distal(), tales_compare_funclal(), tales_domain_distances(), tales_group_kmedoids(), tales_tale_distances(), validate_pairwise_distances()]`

Examples

```
x <- tales_from_telltales(system.file("extdata", "tellTaleExampleOutput",
                                     package = "tantale"))

cmp <- tales_compare_distal(x)
grouped <- tales_group_hclust(cmp$tales, cmp$tale_distances, k = 2)
unique(grouped[c("array_id", "group")])
```

<code>tales_group_kmedoids</code>	<i>Group TALEs by k-medoids clustering of their pairwise distance</i>
-----------------------------------	---

Description

Clusters TALE arrays with `pam` for every candidate `k` in `k_range`, then returns the `tales` object the distances were computed from with the chosen clustering's result attached as its group column.

Usage

```
tales_group_kmedoids(
  x,
  tale_distances,
  k_range = NULL,
  k = NULL,
  seed = 7,
  plot_silhouette = TRUE
)
```

Arguments

<code>x</code>	A <code>tales</code> object – the one whose comparison produced <code>tale_distances</code> .
<code>tale_distances</code>	A <code>tale_distances</code> object, as returned by <code>tales_compare_distal()</code> . A plain data frame with the same columns is also accepted and coerced.
<code>k_range</code>	Integer vector of candidate values of <code>k</code> to evaluate. Each must be between 1 and one less than the number of arrays being grouped (<code>cluster::pam()</code> 's own requirement).
<code>k</code>	See Details.
<code>seed</code>	Passed to <code>set.seed()</code> before every <code>cluster::pam()</code> call, so the same candidate always clusters the same way.
<code>plot_silhouette</code>	Logical, whether to draw the silhouette-vs-k plot.

Details

The function clusters `tale_distances` but returns the `tales` object those distances came from, with the groups added. `group` is a recognised `tales` column, validated as constant within an array – it is an array-level property, like `seqnames`.

Taking `x` rather than returning a bare lookup table is what makes the correspondence checkable: the array names in `tale_distances` must be the array names in `x`, and this is the only place that can be verified. A mismatch is an error rather than a silent NA group, because a partly-grouped object is the kind of thing that fails much later and confusingly.

The bare mapping is still one line away if you want it: `unique(out[c("array_id", "group")])`.

Unlike a hierarchical tree, PAM has no single structure that can be cut at an arbitrary `k` after the fact: `k` is a parameter of the clustering itself. So one clustering is computed per candidate in `k_range`, and `k` picks which of those to keep:

- `k` a single number uses that candidate directly.
- `k = "auto"` picks the elbow of the silhouette-vs-`k` curve, the point after which adding groups stops improving the fit much. It uses the first step of the Kneedle algorithm only, a practical heuristic; check the silhouette plot when the choice matters.
- `k = NULL` (the default) shows the silhouette plot and asks for a number at the console, but only when `interactive()` is TRUE. In a script, a test or a vignette render, `k = NULL` errors instead of blocking on input that will never arrive.

Value

`x` with an added (or replaced) `group` column.

See Also

`tales_group_hclust()`, the alternative method; `tales_compare_distal()`, which produces both inputs.

Other pairwise distances: `[.pairwise_distances(), as.matrix.pairwise_distances(), distances_assert_square(), distances_restrict(), is_pairwise_distances(), pairwise_distances(), rvd_dna_specificity, tales_assign_domain_codes(), tales_compare_distal(), tales_compare_funcal(), tales_domain_distances(), tales_group_hclust(), tales_tale_distances(), validate_pairwise_distances()]`

Examples

```
x <- tales_from_telltales(system.file("extdata", "tellTaleExampleOutput",
                                     package = "tantale"))

cmp <- tales_compare_distal(x)
grouped <- tales_group_kmedoids(cmp$tales, cmp$tale_distances,
                               k_range = 2:3, k = 2)
unique(grouped[c("array_id", "group")])
```

tales_msa	<i>Create a tales_msa object</i>
-----------	----------------------------------

Description

A [tales](#) object plus an alignment_position column. Gaps are implicit: a gap is simply the absence of a row at that (array_id, alignment_position).

Usage

```
tales_msa(x, alignment_width = NULL, dom_code_namespace = NULL)
```

Arguments

x A data frame with the [tales](#) columns plus alignment_position.
alignment_width Integer alignment width; defaults to max(alignment_position).
dom_code_namespace Optional scalar string, see [tales_namespace](#).

Details

Normally produced by [tales_align](#) rather than called directly.

Value

A validated tales_msa object.

See Also

Other TALE alignment: [as.matrix.tales_msa\(\)](#), [is_tales_msa\(\)](#), [tales_align\(\)](#), [tales_consensus\(\)](#), [tales_consensus_match\(\)](#), [tales_msa_params\(\)](#), [tales_msa_width\(\)](#), [validate_tales_msa\(\)](#)

Examples

```
# A hand-built alignment: A2 has a gap at position 2 relative to A1.
aligned <- data.frame(
  array_id = c("A1", "A1", "A1", "A2", "A2"),
  position_in_array = c(1L, 2L, 3L, 1L, 2L),
  alignment_position = c(1L, 2L, 3L, 1L, 3L),
  rvd = c("NTERM", "HD", "CTERM", "NTERM", "CTERM")
)
msa <- tales_msa(aligned)
tales_msa_width(msa)
as.matrix(msa)
```

tales_msa_params	<i>Settings a TALE alignment was made with</i>
------------------	--

Description

What [tales_align](#) aligned on and how, kept with the alignment so that it can be reproduced or interpreted later, for example to tell whether a `tales_msa` holding both `rvd` and `dom_code` columns was aligned on its RVDs or on its domain codes. Subsetting keeps these settings; returning to a plain `tales` ([as_tales](#)) drops them.

Usage

```
tales_msa_params(x)
```

Arguments

`x` A `tales_msa` object.

Value

A list with `residue_col`, the column aligned on; `domain_distances`, the scoring given to `tales_align()` (NULL, "rvd", or the matrix, table or file supplied); and `mafft_opts`, the options MAFFT ran with. NULL for a `tales_msa` built by hand with [tales_msa](#).

See Also

Other TALE alignment: [as.matrix.tales_msa\(\)](#), [is_tales_msa\(\)](#), [tales_align\(\)](#), [tales_consensus\(\)](#), [tales_consensus_match\(\)](#), [tales_msa\(\)](#), [tales_msa_width\(\)](#), [validate_tales_msa\(\)](#)

Examples

```
# Needs MAFFT, resolved from the tantale conda environment on first use.
rvd_fasta <- system.file("extdata", "TalA_RVDSeqs_AnnoTALE.fasta",
                        package = "tantale")
msa <- tales_align(as_tales(rvd_fasta, sep = "-"), residue_col = "rvd")
tales_msa_params(msa)$residue_col
```

tales_msa_width	<i>Width of a TALE alignment</i>
-----------------	----------------------------------

Description

The number of columns in the alignment, including those that are all gaps in the object at hand. It is stored with the object, because subsetting arrays can empty the last column, which would silently shrink `max(alignment_position)`. A plain tales has no width.

Usage

```
tales_msa_width(x)
```

Arguments

`x` A tales_msa object.

Value

An integer scalar, or NULL if unset.

See Also

Other TALE alignment: [as.matrix.tales_msa\(\)](#), [is_tales_msa\(\)](#), [tales_align\(\)](#), [tales_consensus\(\)](#), [tales_consensus_match\(\)](#), [tales_msa\(\)](#), [tales_msa_params\(\)](#), [validate_tales_msa\(\)](#)

Examples

```
# A2 has no C-terminus, so the last column holds A1's alone.
aligned <- data.frame(
  array_id = c("A1", "A1", "A1", "A1", "A2", "A2"),
  position_in_array = c(1L, 2L, 3L, 4L, 1L, 2L),
  alignment_position = c(1L, 2L, 3L, 4L, 1L, 2L),
  rvd = c("NTERM", "HD", "NI", "CTERM", "NTERM", "HD")
)
msa <- tales_msa(aligned)
tales_msa_width(msa)
# Keeping A2 alone empties columns 3 and 4; the width is still 4.
a2 <- msa[msa$array_id == "A2", ]
max(a2$alignment_position)
tales_msa_width(a2)
```

tales_names*The names of the TALEs in a tales object*

Description

Each array in a `tales` object is one TALE, identified by its `array_id`. This returns those identifiers once each, in the order the arrays appear in `x`. They are also the names of the vector that `tales_rvd_strings` returns. A `tales_msa` has the same arrays as the `tales` object it was aligned from.

Usage

```
tales_names(x)
```

Arguments

`x` A `tales` object.

Details

`names()` is left to its usual meaning: a `tales` object is a data frame, so `names(x)` returns its column names.

Value

A character vector of array identifiers.

See Also

Other `tales` objects: `[.tales()`, `as_tales()`, `format.tales()`, `format.tales_msa()`, `is_tales()`, `new_tales()`, `print.tales()`, `print.tales_msa()`, `summary.tales()`, `summary.tales_msa()`, `tales()`, `tales_anchor_codes()`, `tales_anomalies()`, `tales_assert_complete()`, `tales_bind()`, `tales_namespace()`, `tales_requirements()`, `validate_tales()`

Examples

```
rvd_fasta <- system.file("extdata", "TalA_RVDSeqs_AnnoTALE.fasta",
                          package = "tantale")
x <- as_tales(rvd_fasta, sep = "-")
tales_names(x)
length(tales_names(x)) # the number of TALEs
```

tales_namespace	<i>The dom_code namespace of a tales object</i>
-----------------	---

Description

Identifies the run whose dom_code values this object carries. Objects from different runs must not be joined: dom_code is minted per run, so the codes collide across runs and a join would silently succeed against the wrong domains.

Usage

```
tales_namespace(x)
```

Arguments

x A tales object.

Value

A scalar string, or NULL if the object is not stamped.

See Also

Other tales objects: [\[.tales\(\)\]](#), [as_tales\(\)](#), [format.tales\(\)](#), [format.tales_msa\(\)](#), [is_tales\(\)](#), [new_tales\(\)](#), [print.tales\(\)](#), [print.tales_msa\(\)](#), [summary.tales\(\)](#), [summary.tales_msa\(\)](#), [tales\(\)](#), [tales_anchor_codes\(\)](#), [tales_anomalies\(\)](#), [tales_assert_complete\(\)](#), [tales_bind\(\)](#), [tales_names\(\)](#), [tales_requirements\(\)](#), [validate_tales\(\)](#)

Examples

```
rvd_fasta <- system.file("extdata", "TalA_RVDSeqs_AnnoTALE.fasta",
                        package = "tantale")
x <- as_tales(rvd_fasta, sep = "-")
tales_namespace(x) # NULL -- dom_code has not been minted yet
```

tales_predict_targets	<i>Predict TALE target boxes</i>
-----------------------	----------------------------------

Description

Runs a TALE target (EBE) prediction tool over a set of TALE RVD sequences and a set of DNA sequences, and returns the predictions as a tibble.

Usage

```
tales_predict_targets(x, subj_file, method = c("talvez", "preditale"), ...)
```

Arguments

<code>x</code>	TALE RVD sequences: a tales object, the path to a fasta file, or a <code>BStringSet</code> . A <code>tales</code> is rendered with tales_rvd_strings , which drops the termini.
<code>subj_file</code>	Path to a fasta file of subject DNA sequence(s).
<code>method</code>	Which backend to use, "talvez" (default) or "preditale".
<code>...</code>	Passed to the chosen backend. See talvez (<code>opt_param</code> , <code>talvez_dir</code> , <code>conda_bin</code>) and preditale (<code>opt_param</code> , <code>java_args</code> , <code>predictor_path</code>); both accept <code>output_dir</code> . Note the two take different <code>opt_param</code> defaults, since the options are the tools' own.

Details

`talvez` and `preditale` are independent programs, but the package normalises their outputs to a shared set of column names, so they can serve as interchangeable backends of one operation. This is that operation; [talvez](#) and [preditale](#) remain available and documented individually, and are where each tool's own options and citation live.

Value

A tibble of EBE predictions with the columns both backends share (see [talvez](#)), then `method`, which tool produced them, then the tool's own column (`rank` for `Talvez`, `pval` for `PrediTALE`).

See Also

[talvez](#), [preditale](#)

Other target prediction: [plot_target_preds\(\)](#), [preditale\(\)](#), [talvez\(\)](#)

Examples

```
# Needs the tantale conda environment (for talvez) and a Java runtime
# (for preditale), both built on first use.
x <- tales_from_telltales(system.file("extdata", "tellTaleExampleOutput",
                                     package = "tantale"))
subj <- system.file("extdata", "cladeIII_sweet_promoters.fasta",
                   package = "tantale")
head(tales_predict_targets(x, subj_file = subj))
head(tales_predict_targets(x, subj_file = subj, method = "preditale"))
```

tales_requirements

Column requirements of the tales consumers

Description

What each function needs of a [tales](#) object. This is distinct from the class's own column contract, which says what makes an object *valid*: a valid `tales` may still lack what a given operation needs.

Usage

```
tales_requirements()
```

Details

```
tales_compare_distal() aa_seq or dna_seq (translated)
tales_align() whichever of rvd / dom_code residue_col names
tales_rvd_strings() rvd
tales_coded_strings() dom_code
tales_domain_codes() dom_code and aa_seq; rvd included when present
plot() on a tales rvd, aa_seq, domain_type; seqnames adds a facet, alignment_position en-
ables the aligned layout
```

Value

A tibble of function, requirement kind, and columns.

See Also

Other tales objects: [\[.tales\(\)](#), [as_tales\(\)](#), [format.tales\(\)](#), [format.tales_msa\(\)](#), [is_tales\(\)](#), [new_tales\(\)](#), [print.tales\(\)](#), [print.tales_msa\(\)](#), [summary.tales\(\)](#), [summary.tales_msa\(\)](#), [tales\(\)](#), [tales_anchor_codes\(\)](#), [tales_anomalies\(\)](#), [tales_assert_complete\(\)](#), [tales_bind\(\)](#), [tales_names\(\)](#), [tales_namespace\(\)](#), [validate_tales\(\)](#)

Examples

```
tales_requirements()
```

tales_rvd_strings	<i>RVD strings, one per TALE array</i>
-------------------	--

Description

Renders each array as a separated string of its RVDs in part order — the form target-prediction tools consume.

Usage

```
tales_rvd_strings(x, sep = "-", repeats_only = TRUE)
```

Arguments

x	A tales object carrying an rvd column.
sep	Separator between RVDs. Defaults to "-", the convention used by AnnoTALE and by this package's own sample files.
repeats_only	Drop the terminus parts, whose rvd holds an anchor code rather than a real RVD (see tales_anchor_codes). TRUE by default, since target prediction concerns the central repeat domain only.

Value

A [BStringSet](#), named by array_id.

See Also

[tales_coded_strings](#)

Other tales projections: [tales_coded_strings\(\)](#), [tales_domain_codes\(\)](#), [tales_get_dna_seq\(\)](#), [tales_get_protein_seq\(\)](#), [tales_to_universalmotif\(\)](#)

Examples

```
x <- tales_from_telltales(system.file("extdata", "tellTaleExampleOutput",
                                     package = "tantale"))
tales_rvd_strings(x)[1]
tales_rvd_strings(x, repeats_only = FALSE)[1] # keeps NTERM/CTERM markers
```

tales_tale_distances	<i>Pairwise distances between TALE arrays</i>
----------------------	---

Description

Step 3 of [tales_compare_distal\(\)](#). Aligns the arrays against each other as strings of domain codes, using the domain distances from step 2 as the cost of substituting one domain for another.

Usage

```
tales_tale_distances(x, domain_distances)
```

Arguments

x A [tales](#) object carrying dom_code.

domain_distances

A [domain_distances](#) object for the same x, as returned by [tales_domain_distances\(\)](#).

Details

The two arguments are not independent, and that is the point. Each array's sequence of dom_codes is aligned against every other's; what it costs to align one domain against a different one is taken from domain_distances, so the TALE-level comparison is built on the domain-level one.

The alignment is the minisatellite map alignment of ARLEM (Abouelhoda, Giegerich, Behzadi and Steyaert, 2009), which DisTAL used because repeat arrays evolve like minisatellites: besides substitutions, a run of domains that one array has and the other lacks can be explained as tandem duplications of a neighbouring domain, each copy then free to diverge. tantale computes it in R with the costs DisTAL gave ARLEM: a duplication costs 10 plus the substitution cost between the copy and its source, and a domain inserted from elsewhere costs 10. The scores are identical to those of the original ARLEM program. arlem_score is the cost of the best alignment, and dissim divides it by the length of the longer array.

The domain distances are first passed through a Minkowski distance ($p = 3.5$) between their rows and rescaled to 0-100. That step matters: the alignment needs substitution costs satisfying the triangle inequality, and raw pairwise alignment dissimilarities do not.

Value

A `tales_distances` object, keyed by `array_id`.

Both arguments must come from the same call

`domain_distances` is keyed by `dom_code`, and those codes mean what they mean only within the call that minted them (`tales_assign_domain_codes()`). Passing distances from one run with codes from another silently compares the wrong domains, which is why both objects carry a namespace stamp and this function refuses when they disagree.

See Also

`tales_compare_distal()`, which runs all three steps.

Other pairwise distances: `[.pairwise_distances(), as.matrix.pairwise_distances(), distances_assert_square(), distances_restrict(), is_pairwise_distances(), pairwise_distances(), rvd_dna_specificity, tales_assign_domain_codes(), tales_compare_distal(), tales_compare_funcal(), tales_domain_distances(), tales_group_hclust(), tales_group_kmedoids(), validate_pairwise_distances()]`

Examples

```
x <- tales_from_telltales(system.file("extdata", "tellTaleExampleOutput",
                                     package = "tantale"))
xa <- tales_assign_domain_codes(x)
dd <- tales_domain_distances(xa)
tales_tale_distances(xa, dd)
```

tales_to_universalmotif

Convert a tales object to a list of predicted DNA-binding-specificity PWMs

Description

One `universalmotif` position weight matrix per array, built by looking up each repeat's RVD in `rvd_dna_specificity` and stacking the rows in repeat order. The conversion `tales_compare_funcal` is built on, exposed on its own so any `universalmotif` function (`compare_motifs()`, `motif_tree()`, `view_motifs()`, `scan_sequences()`, `merge_motifs()`, ...) can be run directly on these TALE binding models.

Usage

```
tales_to_universalmotif(x)
```

Arguments

`x` A [tales](#) object carrying an rvd column.

Details

Only rvd, in repeat order, is used (via [tales_rvd_strings](#), which drops the two termini by default): DNA-binding specificity is a property of the repeat region, and QueTAL's FuncTAL, the tool this table comes from, scored repeats only. An RVD absent from [rvd_dna_specificity](#) is given a flat, uninformative row, so an unusual RVD lowers the specificity of the comparison without causing an error.

Value

A named list of [universalmotif](#) objects, one per array, named by `array_id`: a plain list, which is what `compare_motifs()`/`motif_tree()` accept.

See Also

[tales_compare_funcTAL\(\)](#), the one comparison built on this; [tales_rvd_strings\(\)](#), the sibling projection at the string layer.

Other tales projections: [tales_coded_strings\(\)](#), [tales_domain_codes\(\)](#), [tales_get_dna_seq\(\)](#), [tales_get_protein_seq\(\)](#), [tales_rvd_strings\(\)](#)

Examples

```
x <- tales_from_telltales(system.file("extdata", "tellTaleExampleOutput",
                                     package = "tantale"))
motifs <- tales_to_universalmotif(x)
motifs[[1]]
```

talomes_heatmap

Heatmap of RVD sequence variants across strains and TALE groups

Description

Draws a talome overview: one column per TALE group, one row per strain, each cell showing which RVD sequence variant that strain carries in that group. A strain's talome is its whole complement of TALEs, so the plot shows at a glance which groups each strain has and where strains carry different variants of the same TALE.

Within a group, the cell colour gives each variant's rank by the number of strains that carry it: the most common variant is the palest and rarer ones are darker. The # after each group label counts its distinct variants. A white cell means the strain has no member in that group. A cell can hold several colours side by side when a strain carries more than one variant in the same group. Dendrograms order strains and groups by the similarity of their variant profiles.

Usage

```
talomes_heatmap(
  tale_annotation,
  group_col = "group",
  strain_col,
  rvd_col,
  trunc_tales_col = NULL,
  extra_col = NULL,
  x_lab = "TALE Group",
  y_lab = "Strain",
  title = "RVD sequences variants",
  plot_type = "all",
  colors = .tantale_colours$variant_ranks,
  margins = NULL,
  sep_width = 5,
  sep_color = "white",
  inner_sep_color = "white",
  save_path = NULL
)
```

Arguments

<code>tale_annotation</code>	A data frame with one row per TALE and at least a group, a strain and an RVD-sequence column. Or a tales object carrying group and strain columns, with one value per array (for instance a grouped tales to which a strain column was added): the RVD sequences are then computed with tales_rvd_strings , and arrays without repeats are left out.
<code>group_col</code>	Name of the <code>tale_annotation</code> column holding TALE groups, drawn as columns. The default, "group", is the column tales_group_kmedoids and tales_group_hclust add.
<code>strain_col</code>	Name of the column holding strain names, drawn as rows.
<code>rvd_col</code>	Name of the column holding RVD sequences (e.g. from tales_rvd_strings). Not needed when <code>tale_annotation</code> is a <code>tales</code> object.
<code>trunc_tales_col</code>	Optional name of a logical column marking truncated TALEs; those are labelled "T" in their cell (with <code>plot_type = "all"</code>).
<code>extra_col</code>	Optional name of a column with further information about each strain (e.g. origin), drawn as a side bar on the right.
<code>x_lab, y_lab, title</code>	Axis names and plot title.
<code>plot_type</code>	Either "all" to draw every allele, or "single" to draw one representative allele per group.
<code>colors</code>	Colours from the most common variant to the rarest. They are interpolated over the ranks present, so the most common variant always gets the first colour and the rarest the last. The default runs from pale to dark wine, so a rare variant stands out.

margins	Space for the row dendrogram, column dendrogram, row names and column names, in that order, counted in heatmap cells. NULL (default) sizes the dendrograms to a quarter of the heatmap's width and height, between 1.5 and 5 cells. With <code>plot_type = "all"</code> , it gives the names and the title the room their text takes, whatever the size of the device, and the cells share the rest; with <code>plot_type = "single"</code> , the names get 3 cells, and the column dendrogram also holds the title and gets at least 3.5 cells.
sep_width	Width of the separator between adjacent cells.
sep_color	Colour of the separator between adjacent cells.
inner_sep_color	Colour of the separator between variants within one cell.
save_path	Optional file path; the image format follows the file extension. If NULL (default), the heatmap is drawn on the current device.

Value

NULL, invisibly. Called for the side effect of drawing the heatmap (or, if `save_path` is given, writing it to a file).

See Also

Other TALE plots: `plot.tales()`, `plot.tales_msa()`, `plot_target_preds()`

Examples

```
ann <- data.frame(
  group = c("G1", "G1", "G1", "G2", "G2"),
  strain = c("S1", "S2", "S3", "S1", "S2"),
  rvdseq = c("NI-HD-NG", "NI-HD-NG", "NN-HD-NG",
            "HD-NI-NG-NG", "HD-NI-NG-NG")
)
talomes_heatmap(ann, group_col = "group", strain_col = "strain",
               rvd_col = "rvdseq")

# From a tales object: one group and one strain per array
x <- tales_from_telltales(system.file("extdata", "tellTaleExampleOutput",
                                     package = "tantale"))
x$group <- c(ROI_00001 = 1, ROI_00002 = 1, ROI_00003 = 2, ROI_00004 = 2)[x$array_id]
x$strain <- c(ROI_00001 = "S1", ROI_00002 = "S2", ROI_00003 = "S1",
             ROI_00004 = "S2")[x$array_id]
talomes_heatmap(x, strain_col = "strain")
```

`talvez`

Run TALE target predictions on DNA sequence(s) using Talvez

Description

An R wrapper around the **Talvez** predictor Perl script. Takes a list of TALE RVD sequences and a fasta file of DNA sequences and runs Talvez.

Usage

```
talvez(
  rvd_seqs,
  subj_file,
  opt_param = "-t 0 -l 19",
  output_dir = NULL,
  talvez_dir = system.file("tools", "TALVEZ_3.2", package = "tantale", mustWork = T),
  conda_bin = "auto"
)
```

Arguments

<code>rvd_seqs</code>	Tale RVD sequences are supplied as either a fasta file (atomic character vector) with Tale info (name) in title and sequences of RVD as a space or '-' separated string or as a Biostrings XStringSet with sequences of RVD similarly formatted.
<code>subj_file</code>	Expects a character vector specifying the path to the fasta file holding subject DNA sequence(s). Talvez does not accept sequences wrapped at a fixed width. The function unwraps them with Biostrings, but this fails for subject sequences longer than 20 kb: unwrap such sequences beforehand.
<code>opt_param</code>	An atomic character vector specifying optional parameters for the Talvez script (eg "-t 0 -l 19"). These may not include the '-e' and '-z' options specifying the matrix files.
<code>output_dir</code>	Expects a character vector specifying the path to an output directory. If not supplied, output files will be temporary.
<code>talvez_dir</code>	If you want to use another version of Talvez than the one supplied with tantale, specify the path of the directory containing the necessary files here.
<code>conda_bin</code>	Path to your Conda binary file if you need to specify a path different from the one that is automatically searched by the reticulate package functions.

Details

This wrapper uses the RVD-nucleotide specificity matrices shipped with Talvez; custom matrices are not supported. The Talvez script runs in a conda environment providing its dependencies, created automatically on first use.

Value

A tibble with one row per predicted EBE (effector binding element), the columns renamed from the tool's own output so that `talvez` and `preditale` return the same ones in the same order: `tale_id`, `rvds`, `subj_seq_id`, `start`, `end`, `strand`, `ebe_seq` and `score`, followed by the tool's own column: `rank`, the site's rank among the TALE's predictions by score.

See Also

Other target prediction: `plot_target_preds()`, `preditale()`, `tales_predict_targets()`

Examples

```
# Needs the tantale conda environment, built on first use.
x <- tales_from_telltales(system.file("extdata", "tellTaleExampleOutput",
                                     package = "tantale"))

rvds <- tales_rvd_strings(x)
subj <- system.file("extdata", "cladeIII_sweet_promoters.fasta",
                  package = "tantale")
preds <- talvez(rvd_seqs = rvds, subj_file = subj)
head(preds)
```

tantale_genome	<i>Path to one of the example genomes</i>
----------------	---

Description

The articles and examples use four *Xanthomonas oryzae* pv. *oryzae* genome assemblies. They are not part of the package itself: [tantale_setup\(\)](#) downloads them, with the Java tools, when called with `install = TRUE`. This function returns where one of them is.

Usage

```
tantale_genome(strain = c("MAI1", "BAI3", "BAI3-1-1", "PX086"))
```

Arguments

strain	Which genome.
--------	---------------

Details

MAI1 (GenBank CP025609.1), BAI3 (CP025610.1) and PX086 (RefSeq NZ_CP007166.1) are complete genomes of African and Asian strains; the sequences are those of the records, with shortened FASTA headers. BAI3-1-1 is an unpublished assembly of a BAI3 derivative that carries sequencing and assembly errors in its TALE loci. It was produced by the authors of tantale, is available only from the package's genomes-1 release, and serves to illustrate frameshift correction.

Value

The path of a FASTA file.

See Also

[tantale_setup\(\)](#), which downloads them.

Examples

```
# Needs the genomes downloaded by tantale_setup(install = TRUE).
tantale_genome("MAI1")
```

tantale_setup

*Check, and optionally build, tantale's external dependencies***Description**

Reports whether the external programs tantale drives are present and at the versions it expects, and can build or repair the conda environment that provides most of them. It also downloads the Java programs tantale wraps (AnnoTALE, PrediTALE, TALEcorrection) and the four example genomes of the articles ([tantale_genome\(\)](#)), which are too large to be part of the package.

Called bare it changes nothing – it is a diagnostic. Pass `install = TRUE` to act on what it finds.

Usage

```
tantale_setup(
  install = FALSE,
  conda = FALSE,
  conda_bin = "auto",
  archive_dir = NULL
)
```

Arguments

<code>install</code>	Build the tantale environment if it is missing, and repair it if a pinned version is wrong. FALSE by default, so the function reports and changes nothing.
<code>conda</code>	Install a conda distribution if none is found. FALSE by default; see the section above.
<code>conda_bin</code>	Passed to reticulate. "auto" lets it choose.
<code>archive_dir</code>	A directory holding the archives <code>tantale-tools-1.tar.gz</code> and <code>tantale-genomes-1.tar.gz</code> , downloaded beforehand, to install them from instead of downloading them: for a machine without internet access, or to download once for several machines. NULL (default) downloads them.

Details

Why versions are checked as well as presence. MAFFT changed its `--text` mode gap handling after 7.4x, and later versions align TALE repeat-code strings differently, leaving the N- and C-termini unanchored. An environment built by an older version of this package can therefore produce different alignments from the same input, with nothing to indicate it. The pins in `tantale_conda_env.yaml` exist for that reason, and this function checks them.

Why three paths are reported. The conda binary, its default root, and the environment actually in use are three different things, and on a machine with any history they diverge – reticulate scans several known locations, so two roots can each hold an environment named tantale. When that happens, a rebuild can honestly report success while the package goes on using the other one. Everything here therefore works with the environment's prefix (its path), never its name.

Where the downloads go. The function checks two archives, attached to releases of tantale's GitHub repository, against a sha256 recorded in the package and unpacks them into `tools::R_user_dir("tantale",`

"data") (~/.local/share/R/tantale on Linux), in one folder per archive version (tools-1/, genomes-1/). Set the environment variable TANTALE_DATA_DIR to use another directory, for instance one shared by a team. The tools take about 60 MB, the genomes 20 MB.

Java is checked too. It is a hard requirement of the AnnoTALE, PrediTALE and TALE-correction wrappers, it comes from outside conda, and without this check they fail deep inside a system() call.

Value

Invisibly, a list with conda, system and archives data frames of the checks, and prefix, so the result can be tested as well as read. conda is NULL and prefix is NA when no conda/mamba installation, or no tantale environment, was found at all; system is always a data frame, since Java is checked regardless.

Installing conda itself

conda = TRUE installs a conda distribution if none is found. It is deliberately opt-in and separate from install: putting a package manager on someone's machine is a larger side effect than building an environment in one that already exists. It installs **miniconda**, via `reticulate::install_miniconda()`.

See Also

`tell_tales()`, `tales_align()`, `tales_compare_distal()`, `talvez()` and `correct_tales()`, the entry points that need these tools.

Examples

```
# A read-only diagnostic report; nothing is installed or changed.
tantale_setup()
```

tell_tales	<i>Search and report on the features of TALE protein domains potentially encoded in subject DNA sequences</i>
------------	---

Description

tell_tales has been primarily written to report on 'corrected' TALE RVD sequences in indels prone, noisy DNA sequences (suboptimally polished genomes assembly, raw reads of long read sequencing technologies such as PacBio or ONT) that may otherwise be missed by conventional tools (eg AnnoTALE).

Usage

```

tell_tales(
  subject_file,
  output_dir = tempfile("tell_tales_"),
  hmm_dir = system.file("extdata", "hmmProfile", package = "tantale", mustWork = T),
  nterm_min_score = 300,
  repeat_min_score = 20,
  cterm_min_score = 200,
  terminus_max_evalue = 1e-05,
  terminus_min_cover = 0.9,
  min_dna_hits = 4,
  min_array_length = 0,
  min_gap = 35,
  extremity_codes = TRUE,
  rvd_sep = "-",
  hmmer_path = NULL,
  extend_len = 300,
  correct_array = FALSE,
  correction_ref = system.file("extdata", "tale_correction_ref.fa.gz", package =
    "tantale", mustWork = T),
  max_comparisons = 50,
  frameshift = -11,
  ...
)

```

Arguments

subject_file	Fasta file with DNA sequence(s) to be searched for the presence of TALE coding sequences (CDS).
output_dir	Path of the output directory, created if it does not exist. The default is a new directory under <code>tempdir()</code> , which R deletes when the session ends: give a path to keep the results. The directory is returned, so <code>tales_from_telltales(tell_tales(genome))</code> reads the run straight back.
hmm_dir	Folder holding the profile HMMs, if you do not want the ones provided with tantale. It must hold files with the same names: the three DNA profiles of the nhmmer search (<code>Xo_TALE_Nterm_CDS_profile.hmm</code> , <code>Xo_TALE_repeat_CDS_profile.hmm</code> , <code>Xo_TALE_Cterm_CDS_profile.hmm</code>) and the two protein profiles of the terminus check (<code>Xo_TALE_Nterm_AA_profile.hmm</code> , <code>Xo_TALE_Cterm_AA_profile.hmm</code>).
nterm_min_score	Minimal nhmmer score cut_off value to consider the DNA hit as genuine
repeat_min_score	Minimal nhmmer score cut_off value to consider the DNA hit as genuine
cterm_min_score	Minimal nhmmer score cut_off value to consider the DNA hit as genuine
terminus_max_evalue, terminus_min_cover	What it takes for the segment AnnoTALE reports on either side of the repeats to be flagged as a canonical TALE N- or C-terminus, NTERM or CTERM rather than

	XXXXX (see tales_anchor_codes). The segment is searched with the TALE terminal-domain protein profile of <code>hmm_dir</code> ; the match must have an E-value of at most <code>terminus_max_evalue</code> , cover at least <code>terminus_min_cover</code> of the profile's positions, and reach, within 10 positions, the end of the profile that adjoins the repeats. The default of 0.9 comes from the termini of the curated TALEs of tale_annotations : their canonical termini cover 0.93 of the profile or more, the truncated termini of the truncTALEs 0.84 or less.
<code>min_dna_hits</code>	Minimum number of nhmmer hits for a subject sequence (a contig, a chromosome) to be considered further. A cheap way to discard whole sequences that carry nothing but stray matches, before any expensive work is done on them. It says nothing about the length of the TALE arrays found within a sequence that passes – see <code>min_array_length</code> for that.
<code>min_array_length</code>	Minimum number of repeat units for a TALE array to be kept. Defaults to 0, which keeps everything. Counting repeats rather than all hits means an array is not penalised for having had its termini missed, and matches what "array length" usually means for a TALE: the number of repeats is what determines how long a target box it recognises. Whether a short array is noise or a genuinely truncated TALE is a judgement about the biology, which is why nothing is discarded unless you ask. A pseudogene with three surviving repeats is real, and may be what you are looking for.
<code>min_gap</code>	Minimum gap in base pairs between two tale domain DNA hits for them to be considered distinct. If the length of the gap is below this value, domains are considered "contiguous" and grouped in the same array.
<code>extremity_codes</code>	Set this to FALSE if you do not want the terminus codes in the RVD strings of <code>rvd_sequences.fas</code> and <code>array_report.tsv</code> .
<code>rvd_sep</code>	Symbol acting as a separator in RVD sequences
<code>hmmmer_path</code>	Specify the path to a directory holding the HMMER executable if you do not want to use the ones provided with tantale.
<code>extend_len</code>	number of nucleotides to extend in 3'-end at the tal ORF prediction stage.
<code>correct_array</code>	Whether to pass each array through CorrectFrameshifts before AnnoTALE sees it. FALSE by default; see Details for what turning it on buys (removing artefactual indels) and risks (erroneously "correcting" a genuine frameshift).
<code>correction_ref</code>	Fasta of reference TALE proteins to correct against. Two are shipped, both built by <code>data-raw/make_correction_references.R</code> from the same source: • <code>tale_correction_ref.fa.gz</code> (default, 494 sequences) – every distinct TALE protein of at least 300 aa found across 70 <i>Xanthomonas oryzae</i> genomes. • <code>tale_correction_ref_representative.fa.gz</code> (136) – a diversity-sampled subset, for a smaller footprint. Both keep the pseudogenes. Their frameshifts came from high-quality genomes and so are assumed real. A reference holding only intact TALEs would "repair" a genuine pseudogene into an ORF that no strain carries.

max_comparisons	How many reference proteins each array may be aligned against during frameshift correction, and the main control on how long correction takes. NULL allows all of them. It is passed to <code>DECIPHER::CorrectFrameshifts()</code> , which ranks the references by a quick distance and aligns the array against the closest max_comparisons of them. The TALE mining article compares values on the genomes shipped with the package.
frameshift	Frameshift penalty passed to <code>CorrectFrameshifts</code> 's <code>frameShift</code> . <code>tantale</code> 's default is -11, overriding <code>DECIPHER</code> 's own -15; change it only with a reason.
...	Additional parameters for the <code>CorrectFrameshifts</code> function.

Details

The approach is first to use [HMMER](#) to find and categorize regions in the input DNA sequence that are related to the coding sequence of canonical TALE protein domains (N-Term, repeats, C-term). It then groups hits that are adjacent, or nearly so (see `min_gap`), into "taleArrays", each a potential *tal* gene.

With `correct_array = FALSE`, `tell_tales()` gives [AnnoTALE](#) the longest open reading frame of each taleArray (extended by `extend_len`) to detect TALE domains in the predicted translation product. The Results should hence be very similar to what would be obtained with AnnoTALE, plus many additional informative output files such as tabular reports.

With `correct_array = TRUE`, `CorrectFrameshifts` first tries to 'correct' potential frameshifts in the taleArray sequences. This conveniently removes many artefactual indels but bear in mind that this may also **erroneously** 'correct' genuine frame shifts which can be highly relevant especially for truncTALEs or iTALEs. The resulting 'corrected' taleArray open reading frames are then passed to AnnoTALE.

Note that occasionally, when a putative open reading frame does not encode a canonical TALE protein (early frame shift, incomplete ORF, etc...), the "analyze" module of AnnoTALE outputs DNA parts but no protein parts and/or RVD sequence. This should be detected and reported in the `tell_tales` log.

Each sequence of `subject_file` is treated as linear. A *tal* gene that spans the junction of a circular molecule (the two ends of an assembled chromosome or plasmid) is cut in two, and is reported, if at all, as two partial arrays at the ends of the sequence. Rotating the sequence so that it starts elsewhere avoids this.

Value

Called for its side effects (writing files). If everything runs smoothly, it invisibly returns the path of the directory where output files were written.

List of output files:

- `all_ranges.gff`: gff file of all Tal arrays detected by HMMer
- `array_report.tsv`: one row per candidate TALE array. Columns named `*_dna_*` describe the nhmmer search of the subject DNA; columns named `*_aa_*` describe the protein segments AnnoTALE extracted from the array's longest ORF.
 - `array_id`, `seqnames`, `start`, `end`, `strand`: the array's identifier and the span of its nhmmer hits on the subject sequence.

- *n_dna_hits*: number of nhmmer hits (N-terminus, repeats and C-terminus profiles together) grouped in the array. A terminus hit usually overlaps the adjacent repeat hit by a few nucleotides, at the boundary between the two domains.
- *array_seq*: DNA sequence of that span.
- *nterm_dna_hit*, *cterm_dna_hit*: whether an nhmmer hit of the N- (C-) terminus DNA profile is part of the array. Only hits scoring at least *nterm_min_score* (*cterm_min_score*) count.
- *rvd_string*: the RVDs AnnoTALE read, separated by *rvd_sep*, with the terminus codes described under *rvd_sequences.fas*. Empty when AnnoTALE found no RVD.
- *has_aberrant_repeat*: whether AnnoTALE flagged a repeat of non-canonical length (a lowercase letter in its RVD).
- *nterm_dna_score*, *cterm_dna_score*, *nterm_dna_evalue*, *cterm_dna_evalue*: bit score and E-value of the best nhmmer hit of the N- (C-) terminus DNA profile next to the array's repeats, whatever its score: on the same strand, upstream of the first repeat (downstream of the last one), within twice the profile length. NA when there is none.
- *nterm_dna_cover*, *cterm_dna_cover*: fraction of the DNA profile's positions those hits cover together; 0 when there is none.
- *nterm_dna_pieces*, *cterm_dna_pieces*: how many such hits. More than one usually means the terminus is split, often by an insertion or a deletion.
- *nterm_aa_evalue*, *cterm_aa_evalue*, *nterm_aa_score*, *cterm_aa_score*: E-value and bit score of the hmmsearch match between the segment AnnoTALE reported upstream (downstream) of the repeats and the TALE N- (C-) terminal protein profile. NA when there is no segment, or no match with an E-value up to 10.
- *nterm_aa_profile_gap*, *cterm_aa_profile_gap*: number of profile positions between the end of that match and the end of the profile that adjoins the repeats (the last position of the N-terminal profile, the first of the C-terminal one). 0 for a match that reaches the repeats, NA when there is no match.
- *nterm_aa_far_gap*, *cterm_aa_far_gap*: the same at the other end of the profile, the start of the N-terminal one (the end of the C-terminal one). A terminus that stops early, like the C-terminus of a truncTALE, leaves a large gap here.
- *nterm_aa_cover*, *cterm_aa_cover*: fraction of the protein profile's positions the match covers. Below 1 minus the two gaps when part of the terminus is missing in between.
- *nterm_aa_domains*, *cterm_aa_domains*: how many separate stretches of the profile the match consists of. More than one means the terminus is split, by a deletion or a change of reading frame.
- *nterm_aa_hit*, *cterm_aa_hit*: TRUE for a canonical terminus: E-value at most *terminus_max_evalue*, cover at least *terminus_min_cover*, profile gap at most 10. FALSE for any other segment, NA when AnnoTALE reported no segment on that side.
- *nterm_aa_length*, *cterm_aa_length*: length of those segments in amino acid residues, excluding a stop codon, as in the *tales* object's *aa_seq*.
- *longest_orf_length*, *longest_orf_seq*: the longest ORF found in the array region extended by *extend_len* nucleotides at its 3' end.
- *orf_coverage*: that ORF's length as a percentage of the extended region's length.
- *predicted_dels_count*, *predicted_ins_count* (with *correct_array* = TRUE): number of putative deletions/insertions in the raw sequence that [CorrectFrameshifts](#) corrected.

- hits_report.tsv: report of all hits detected by HMMer, with their bit *score*, *evaluate* and the profile positions they cover, *hmm_from* to *hmm_to* (for hits merged into one, the best score and E-value and the span they cover together)
- hits_report.gff: gff file of all hits detected by HMMer
- domains_report.tsv: report of all Tal amino acid domains detected by AnnoTALE analyze
- putative_tal_orf.fasta: for each array in which AnnoTALE found RVDs, the DNA of the longest ORF of the array region extended by *extend_len* nucleotides at its 3' end (after frameshift correction with *correct_array* = TRUE). This is the putative TALE coding sequence AnnoTALE analysed.
- pseudo_tal_cds.fasta: for each array in which AnnoTALE found no RVD, the DNA of the array region extended by *extend_len* nucleotides at its 3' end, as found in *subject_file*: candidate pseudogenes, assembly errors or false detections, kept for inspection.
- rvd_sequences.fas: the RVDs (separated by *rvd_sep*) of each array for which AnnoTALE found at least one. With *extremity_codes* = TRUE (the default), each string is bracketed by terminus codes (see [tales_anchor_codes](#)): NTERM (CTERM) when the segment AnnoTALE reported upstream (downstream) of the repeats matches the TALE N- (C-) terminal protein profile, XXXXX when it does not, and no code when AnnoTALE reported no segment on that side.
- c_terminus_aa_alignment.html: protein alignment of all C-termini (only written when at least 2 were found; skipped with a warning otherwise)
- c_terminus_dna_alignment.html: DNA alignment of all C-termini (same condition)
- n_terminus_aa_alignment.html: protein alignment of all N-termini (same condition)
- n_terminus_dna_alignment.html: DNA alignment of all N-termini (same condition)
- tale_cds_all_diagnostic_regions_hmmfile.out: HMMER profile used for tale cds search.
- hmmer_search_out.txt: ignore
- nhmmer_human_readable_output_of_last_run.txt: primary HMMER output file.
- tell_tales.log: a log file
- annotale folder: folder containing result of AnnoTALE analyze for all Tal arrays
- correction_alignment_aa folder: folder containing protein alignment of Tal array detected by HMMer and corrected Tal array if *correct_array* = TRUE
- correction_alignment_dna folder: folder containing DNA alignment of Tal array detected by HMMer and corrected Tal array if *correct_array* = TRUE

See Also

Other TALE discovery: [correct_tales\(\)](#), [tale_annotations](#), [tales_from_annotale\(\)](#), [tales_from_telltales\(\)](#)

Examples

```
# Needs nhmmer and AnnoTALE, resolved from the tantale conda environment
# (and a Java runtime) on first use.
subj <- system.file("extdata", "bai3_sample_tal_genomic_regions.fasta",
                    package = "tantale")
out <- tempfile("tell_tales_example")
tell_tales(subject_file = subj, output_dir = out)
tales_from_telltales(out)
```

`validate_pairwise_distances`*Validate a pairwise distance table*

Description

Checks the column contract, and nothing else. Squareness, the diagonal and symmetry are preconditions of the methods that need them, checked by [distances_assert_square](#): a table filtered on one id column is a legitimate intermediate step.

Usage

```
validate_pairwise_distances(x)
```

Arguments

`x` A pairwise_distances object.

Value

`x`, invisibly, if valid; otherwise an error.

See Also

Other pairwise distances: [\[.pairwise_distances\(\)](#), [as.matrix.pairwise_distances\(\)](#), [distances_assert_square\(\)](#), [distances_restrict\(\)](#), [is_pairwise_distances\(\)](#), [pairwise_distances\(\)](#), [rvd_dna_specificity](#), [tales_assign_domain_codes\(\)](#), [tales_compare_distal\(\)](#), [tales_compare_functal\(\)](#), [tales_domain_distances\(\)](#), [tales_group_hclust\(\)](#), [tales_group_kmedoids\(\)](#), [tales_tale_distances\(\)](#)

Examples

```
d <- data.frame(
  id1 = c("A1", "A1", "A2", "A2"),
  id2 = c("A1", "A2", "A1", "A2"),
  dissim = c(0, 35, 35, 0)
)
validate_pairwise_distances(pairwise_distances(d))

# Without its dissim column, a table is not a pairwise_distances
try(validate_pairwise_distances(d[, c("id1", "id2")]))
```

validate_tales	<i>Validate a tales object</i>
----------------	--------------------------------

Description

Checks the column contract and every invariant that still holds on any subset of rows. Properties that hold only of a *complete* object (an array carrying all its parts, numbered contiguously from 1) are checked by the functions that need them, such as alignment; see [tales_assert_complete](#).

Usage

```
validate_tales(x)
```

Arguments

x A tales object.

Value

x, invisibly, if valid; otherwise an error.

See Also

Other tales objects: [\[.tales\(\)](#), [as_tales\(\)](#), [format.tales\(\)](#), [format.tales_msa\(\)](#), [is_tales\(\)](#), [new_tales\(\)](#), [print.tales\(\)](#), [print.tales_msa\(\)](#), [summary.tales\(\)](#), [summary.tales_msa\(\)](#), [tales\(\)](#), [tales_anchor_codes\(\)](#), [tales_anomalies\(\)](#), [tales_assert_complete\(\)](#), [tales_bind\(\)](#), [tales_names\(\)](#), [tales_namespace\(\)](#), [tales_requirements\(\)](#)

Examples

```
parts <- data.frame(
  array_id = c("A1", "A1"), position_in_array = c(1L, 2L),
  rvd = c("NTERM", "HD")
)
validate_tales(tales(parts))
```

validate_tales_msa	<i>Validate a tales_msa object</i>
--------------------	------------------------------------

Description

Checks every [validate_tales](#) invariant, then those specific to an alignment. As for tales, only properties that still hold on any subset of rows are checked here; grid completeness is a precondition of the functions that need it.

Usage

```
validate_tales_msa(x)
```

Arguments

x A tales_msa object.

Value

x, invisibly, if valid; otherwise an error.

See Also

Other TALE alignment: [as.matrix.tales_msa\(\)](#), [is_tales_msa\(\)](#), [tales_align\(\)](#), [tales_consensus\(\)](#), [tales_consensus_match\(\)](#), [tales_msa\(\)](#), [tales_msa_params\(\)](#), [tales_msa_width\(\)](#)

Examples

```
aligned <- data.frame(
  array_id = c("A1", "A1", "A1", "A2", "A2"),
  position_in_array = c(1L, 2L, 3L, 1L, 2L),
  alignment_position = c(1L, 2L, 3L, 1L, 3L),
  rvd = c("NTERM", "HD", "CTERM", "NTERM", "CTERM")
)
validate_tales_msa(tales_msa(aligned))

# Two parts of A1 placed in the same alignment column
aligned$alignment_position[2] <- 1L
try(tales_msa(aligned))
```

Index

- * **TALE alignment**
 - as.matrix.tales_msa, 9
 - is_tales_msa, 17
 - tales_align, 40
 - tales_consensus, 53
 - tales_consensus_match, 54
 - tales_msa, 64
 - tales_msa_params, 65
 - tales_msa_width, 66
 - validate_tales_msa, 86
- * **TALE discovery**
 - correct_tales, 11
 - tale_annotations, 37
 - tales_from_annotale, 57
 - tales_from_telltales, 58
 - tell_tales, 79
- * **TALE plots**
 - plot.tales, 19
 - plot.tales_msa, 21
 - plot_target_preds, 23
 - talomes_heatmap, 73
- * **datasets**
 - rvd_dna_specificity, 33
 - tale_annotations, 37
- * **external TALE tools**
 - run_annotale_assign, 28
 - run_annotale_build, 29
 - run_annotale_load_classes, 31
 - run_annotale_predict, 32
- * **pairwise distances**
 - [.pairwise_distances, 6
 - as.matrix.pairwise_distances, 8
 - distances_assert_square, 12
 - distances_restrict, 13
 - is_pairwise_distances, 16
 - pairwise_distances, 18
 - rvd_dna_specificity, 33
 - tales_assign_domain_codes, 45
 - tales_compare_distal, 49
 - tales_compare_functal, 51
 - tales_domain_distances, 55
 - tales_group_hclust, 61
 - tales_group_kmedoids, 62
 - tales_tale_distances, 71
 - validate_pairwise_distances, 85
- * **tales objects**
 - [.tales, 7
 - as_tales, 10
 - format.tales, 14
 - format.tales_msa, 15
 - is_tales, 17
 - print.tales, 26
 - print.tales_msa, 27
 - summary.tales, 35
 - summary.tales_msa, 36
 - tales, 39
 - tales_anchor_codes, 42
 - tales_anomalies, 43
 - tales_assert_complete, 44
 - tales_bind, 47
 - tales_names, 67
 - tales_namespace, 68
 - tales_requirements, 69
 - validate_tales, 86
- * **tales projections**
 - tales_coded_strings, 48
 - tales_domain_codes, 55
 - tales_get_dna_seq, 59
 - tales_get_protein_seq, 60
 - tales_rvd_strings, 70
 - tales_to_universalmotif, 72
- * **target prediction**
 - plot_target_preds, 23
 - preditale, 24
 - tales_predict_targets, 68
 - talvez, 75
 - [.pairwise_distances, 6
 - [.pairwise_distances(), 8, 13, 16, 19, 34,

- 46, 50, 52, 56, 62, 63, 72, 85
- [.tales, 7
- [.tales(), 10, 14, 15, 17, 26, 27, 36, 37, 40, 43–45, 48, 67, 68, 70, 86
- aplot, 23
- as.matrix.pairwise_distances, 8
- as.matrix.pairwise_distances(), 6, 13, 16, 19, 34, 46, 50, 52, 56, 62, 63, 72, 85
- as.matrix.tales_msa, 9, 53, 54
- as.matrix.tales_msa(), 18, 41, 53, 54, 64–66, 87
- as_tales, 10, 47, 65
- as_tales(), 7, 14, 15, 17, 26, 27, 36, 37, 40, 43–45, 48, 67, 68, 70, 86
- bind_rows, 47
- Biostrings::AAStringSet, 60
- Biostrings::BStringSet, 49
- Biostrings::DNAStringSet, 59
- BStringSet, 71
- compare_motifs, 51, 52
- correct_tales, 11
- correct_tales(), 4, 39, 58, 59, 79, 84
- CorrectFrameshifts, 81–83
- create_motif, 34
- distances_assert_square, 6, 8, 12, 85
- distances_assert_square(), 6, 8, 13, 16, 19, 34, 46, 50, 52, 56, 62, 63, 72, 85
- distances_restrict, 8, 13
- distances_restrict(), 6, 8, 13, 16, 19, 34, 46, 50, 52, 56, 62, 63, 72, 85
- domain_distances, 4, 41, 50, 56, 71
- domain_distances(pairwise_distances), 18
- dplyr::cur_group_id(), 46
- format.tales, 14, 15, 26
- format.tales(), 7, 10, 15, 17, 26, 27, 36, 37, 40, 43–45, 48, 67, 68, 70, 86
- format.tales_msa, 15, 27
- format.tales_msa(), 7, 10, 14, 17, 26, 27, 36, 37, 40, 43–45, 48, 67, 68, 70, 86
- hclust, 61
- interactive, 63
- is_pairwise_distances, 16
- is_pairwise_distances(), 6, 8, 13, 19, 34, 46, 50, 52, 56, 62, 63, 72, 85
- is_tales, 17
- is_tales(), 7, 10, 14, 15, 26, 27, 36, 37, 40, 43–45, 48, 67, 68, 70, 86
- is_tales_msa, 17
- is_tales_msa(), 9, 41, 53, 54, 64–66, 87
- new_tales(), 7, 10, 14, 15, 17, 26, 27, 36, 37, 40, 43–45, 48, 67, 68, 70, 86
- pairwise_distances, 18
- pairwise_distances(), 6, 8, 13, 16, 34, 46, 50, 52, 56, 62, 63, 72, 85
- pam, 62
- plot.tales, 19
- plot.tales(), 23, 24, 75
- plot.tales_msa, 20, 21
- plot.tales_msa(), 20, 24, 75
- plot_target_preds, 23
- plot_target_preds(), 5, 20, 23, 25, 69, 75, 76
- preditale, 23, 24, 25, 69, 76
- preditale(), 5, 24, 69, 76
- print.summary.tales(summary.tales), 35
- print.summary.tales_msa(summary.tales_msa), 36
- print.tales, 14, 26, 35
- print.tales(), 7, 10, 14, 15, 17, 27, 36, 37, 40, 43–45, 48, 67, 68, 70, 86
- print.tales_msa, 27
- print.tales_msa(), 7, 10, 14, 15, 17, 26, 36, 37, 40, 43–45, 48, 67, 68, 70, 86
- reticulate::install_miniconda(), 5, 79
- run_annotale_assign, 28, 31, 32
- run_annotale_assign(), 4, 30, 32, 33
- run_annotale_build, 28, 29, 29, 32
- run_annotale_build(), 4, 29, 32, 33
- run_annotale_load_classes, 29, 31
- run_annotale_load_classes(), 4, 29, 30, 33
- run_annotale_predict, 28–30, 32, 57
- run_annotale_predict(), 4, 29, 30, 32
- rvd_dna_specificity, 6, 8, 13, 16, 19, 33, 46, 50, 52, 56, 62, 63, 72, 73, 85
- summary(), 56

- summary.tales, 35
- summary.tales(), 7, 10, 14, 15, 17, 26, 27, 37, 40, 43–45, 48, 67, 68, 70, 86
- summary.tales_msa, 36
- summary.tales_msa(), 7, 10, 14, 15, 17, 26, 27, 36, 40, 43–45, 48, 67, 68, 70, 86
- tale_annotations, 4, 12, 37, 58, 59, 81, 84
- tale_distances, 4, 50, 52, 61, 62, 72
- tale_distances(pairwise_distances), 18
- tales, 4, 10, 14, 20, 26, 35, 39, 41, 43, 44, 46–48, 50, 51, 55–62, 64, 69–71, 73, 74
- tales(), 7, 10, 14, 15, 17, 26, 27, 36, 37, 43–45, 48, 67, 68, 70, 86
- tales_align, 34, 40, 64, 65
- tales_align(), 4, 9, 18, 48, 53, 54, 64–66, 79, 87
- tales_anchor_codes, 19, 42, 43, 57, 58, 70, 81, 84
- tales_anchor_codes(), 7, 10, 14, 15, 17, 26, 27, 36, 37, 40, 44, 45, 48, 67, 68, 70, 86
- tales_anomalies, 35, 39, 43, 57, 58
- tales_anomalies(), 4, 7, 10, 14, 15, 17, 26, 27, 36, 37, 40, 43, 45, 48, 67, 68, 70, 86
- tales_assert_complete, 41, 44, 86
- tales_assert_complete(), 7, 10, 14, 15, 17, 26, 27, 36, 37, 40, 43, 44, 48, 67, 68, 70, 86
- tales_assign_domain_codes, 45, 47
- tales_assign_domain_codes(), 6, 8, 13, 16, 19, 34, 50, 52, 56, 62, 63, 72, 85
- tales_bind, 47
- tales_bind(), 7, 10, 14, 15, 17, 26, 27, 36, 37, 40, 43–45, 67, 68, 70, 86
- tales_coded_strings, 48, 55, 71
- tales_coded_strings(), 48, 55, 59, 60, 71, 73
- tales_compare_distal, 20, 21, 35, 38, 47, 49, 51, 52
- tales_compare_distal(), 4, 6, 8, 13, 16, 19, 34, 45, 46, 52, 55, 56, 61–63, 71, 72, 79, 85
- tales_compare_functal, 33, 34, 50, 51, 72
- tales_compare_functal(), 4, 6, 8, 13, 16, 19, 34, 46, 50, 56, 62, 63, 72, 73, 85
- tales_consensus, 22, 36, 53, 54
- tales_consensus(), 4, 9, 18, 41, 54, 64–66, 87
- tales_consensus_match, 54
- tales_consensus_match(), 9, 18, 41, 53, 64–66, 87
- tales_domain_codes, 55
- tales_domain_codes(), 46, 49, 59, 60, 71, 73
- tales_domain_distances, 19, 41, 50, 55
- tales_domain_distances(), 6, 8, 13, 16, 19, 34, 46, 50, 52, 62, 63, 71, 72, 85
- tales_from_annotale, 57
- tales_from_annotale(), 4, 12, 33, 39, 59, 84
- tales_from_telltales, 20, 35, 57, 58
- tales_from_telltales(), 4, 12, 39, 58, 84
- tales_get_dna_seq, 59
- tales_get_dna_seq(), 49, 55, 60, 71, 73
- tales_get_protein_seq, 60
- tales_get_protein_seq(), 49, 55, 59, 71, 73
- tales_group_hclust, 38, 47, 50, 52, 61, 74
- tales_group_hclust(), 4, 6, 8, 13, 16, 19, 34, 46, 50, 52, 56, 63, 72, 85
- tales_group_kmedoids, 47, 50, 52, 62, 74
- tales_group_kmedoids(), 4, 6, 8, 13, 16, 19, 34, 46, 50, 52, 56, 62, 72, 85
- tales_msa, 4, 15, 20, 21, 27, 36, 40, 53, 54, 64, 65
- tales_msa(), 9, 18, 41, 53, 54, 65, 66, 87
- tales_msa_params, 41, 65
- tales_msa_params(), 9, 18, 41, 53, 54, 64, 66, 87
- tales_msa_width, 10, 66
- tales_msa_width(), 9, 18, 41, 53, 54, 64, 65, 87
- tales_names, 67
- tales_names(), 7, 10, 14, 15, 17, 26, 27, 36, 37, 40, 43–45, 48, 68, 70, 86
- tales_namespace, 19, 39, 47, 50, 64, 68
- tales_namespace(), 7, 10, 14, 15, 17, 26, 27, 36, 37, 40, 43–46, 48, 67, 70, 86
- tales_predict_targets, 38, 68
- tales_predict_targets(), 5, 24, 25, 76
- tales_requirements, 69
- tales_requirements(), 7, 10, 14, 15, 17, 26, 27, 36, 37, 40, 43–45, 48, 67, 68, 86
- tales_rvd_strings, 38, 67, 69, 70, 73, 74
- tales_rvd_strings(), 48, 49, 55, 59, 60, 73
- tales_tale_distances, 19, 49, 71

tales_tale_distances(), [6](#), [8](#), [13](#), [16](#), [19](#),
[34](#), [46](#), [48](#), [50](#), [52](#), [56](#), [62](#), [63](#), [85](#)
tales_to_universalmotif, [52](#), [72](#)
tales_to_universalmotif(), [49](#), [52](#), [55](#), [59](#),
[60](#), [71](#)
talomes_heatmap, [38](#), [73](#)
talomes_heatmap(), [4](#), [20](#), [23](#), [24](#)
talvez, [23](#), [25](#), [69](#), [75](#), [76](#)
talvez(), [5](#), [24](#), [25](#), [69](#), [79](#)
tantale (tantale-package), [4](#)
tantale-package, [4](#)
tantale_genome, [77](#)
tantale_genome(), [37](#), [78](#)
tantale_setup, [57](#), [78](#)
tantale_setup(), [5](#), [11](#), [25](#), [29–31](#), [33](#), [77](#)
tell_tales, [11](#), [38](#), [42](#), [57](#), [58](#), [79](#)
tell_tales(), [4](#), [12](#), [37](#), [39](#), [58](#), [59](#), [79](#)
tempdir(), [11](#), [29–31](#), [33](#), [80](#)
tibble, [55](#)

universalmotif, [72](#)

validate_pairwise_distances, [85](#)
validate_pairwise_distances(), [6](#), [8](#), [13](#),
[16](#), [19](#), [34](#), [46](#), [50](#), [52](#), [56](#), [62](#), [63](#), [72](#)
validate_tales, [86](#), [86](#)
validate_tales(), [7](#), [10](#), [14](#), [15](#), [17](#), [26](#), [27](#),
[36](#), [37](#), [40](#), [43–45](#), [48](#), [67](#), [68](#), [70](#)
validate_tales_msa, [86](#)
validate_tales_msa(), [9](#), [18](#), [41](#), [53](#), [54](#),
[64–66](#)