Package: datavolley (via r-universe)

September 7, 2024

Title Reading and Analyzing DataVolley Scout Files

Version 1.8.1

Description Provides functions for parsing and working with volleyball match files in DataVolley format.

Depends R (>= 3.2.4)

URL https://datavolley.openvolley.org,
https://github.com/openvolley/datavolley

BugReports https://github.com/openvolley/datavolley/issues

Imports assertthat, data.table, digest, dplyr, jpeg, jsonlite, lubridate, methods, polyclip, readr, stringi, stringr, vscoututils (>= 0.1.7), xml2

Suggests testthat, ggplot2, knitr, raster, rmarkdown, covr

Encoding UTF-8

License MIT + file LICENSE

RoxygenNote 7.3.1

VignetteBuilder knitr

Remotes openvolley/vscoututils

Repository https://openvolley.r-universe.dev

RemoteUrl https://github.com/openvolley/datavolley

RemoteRef HEAD

RemoteSha a868b26649df5af25a7a3734e1e73b1b983c0695

Contents

`check_player_names` *Check for similar player names*

Description

Player names can sometimes be spelled incorrectly, particularly if there are character encoding issues. This can be a particular problem when combining data from multiple files. This function checks for similar names that might possibly be multiple variants on the same name.

Usage

```
check_player_names(x, distance_threshold = 4)
```

Arguments

`x` datavolley: a datavolley object as returned by `dv_read`, or list of such objects

`distance_threshold` numeric: if two names differ by an amount less than this threshold, they will be returned as possible matches

Value

data.frame

See Also

`dv_read`, `adist`

Examples

```
## Not run:
  x <- dv_read(dv_example_file(), insert_technical_timeouts = FALSE)
  check_player_names(x)

## End(Not run)
```

datavolley **datavolley**

Description

Provides basic functions for parsing Datavolley scout files. Datavolley is a software package used for scouting and summarizing volleyball matches.

Details

The example data files provided with the datavolley package came from `http://www.odbojka.si/`.

Author(s)

Ben Raymond `<ben@untan.gl>`

See Also

Useful links:

- `https://datavolley.openvolley.org`
- `https://github.com/openvolley/datavolley`
- Report bugs at `https://github.com/openvolley/datavolley/issues`

dvlist_summary *Summarize a list of volleyball matches*

Description

Summarize a list of volleyball matches

Usage

```
dvlist_summary(z)
```

Arguments

`z`	list: list of datavolley objects as returned by `dv_read`

Value

named list with various summary indicators, including a competition ladder

See Also

`dv_read`

Examples

```
## Not run:
  x <- dv_read(dv_example_file(), insert_technical_timeouts=FALSE)
 dvlist_summary(list(x,x)) ## same match duplicated twice, just for illustration purposes

## End(Not run)
```

`dv_action2text` *Generate a short, human-readable text summary of one or more actions*

Description

Generate a short, human-readable text summary of one or more actions

Usage

```
dv_action2text(x, verbosity = 1)
```

Arguments

`x`	data.frame or tibble: one or more rows from a datavolleyplays object as returned by `dv_read`
`verbosity`	integer: 1 = least verbose, 2 = more verbose. Currently ignored

Value

character vector

Examples

```
x <- dv_read(dv_example_file())
dv_action2text(plays(x)[27:30, ])
```

`dv_attack_code2desc` *Nominal descriptions for standard attack codes*

Description

Nominal descriptions for standard attack codes

Usage

```
dv_attack_code2desc(code)
```

Arguments

`code` character: vector of attack codes ("X5", "VP", etc)

Value

A named character vector of descriptions. Unrecognized attack codes will have `NA` description.

Examples

```
dv_attack_code2desc(c("X5", "X7", "PP", "blah"))
```

`dv_attack_code2loc` *Nominal starting coordinate for standard attack codes*

Description

Nominal starting coordinate for standard attack codes

Usage

```
dv_attack_code2loc(code)
```

Arguments

`code` character: vector of attack codes ("X5", "VP", etc)

Value

A vector of numeric coordinates

Examples

```
dv_attack_code2loc(code = c("X5", "X7", "PP"))
```

`dv_attack_code2set_type` *Set type for standard attack codes*

Description

Set type for standard attack codes

Usage

```
dv_attack_code2set_type(code)
```

Arguments

`code` character: vector of attack codes ("X5", "VP", etc)

Value

A named vector of sides ("F", "B", "C", "P", "S", "-")

Examples

```
dv_attack_code2set_type(code = c("X5", "X7", "PP"))
```

`dv_attack_code2side` *Attack side for standard attack codes*

Description

Attack side for standard attack codes

Usage

```
dv_attack_code2side(code)
```

Arguments

`code` character: vector of attack codes ("X5", "VP", etc)

Value

A named vector of sides ("L", "R", "C")

Examples

```
dv_attack_code2side(code = c("X5", "X7", "PP"))
```

`dv_attack_code_map` *Translate attack type and starting zone into an attack code.*

Description

If your DataVolley files does not have attack codes ready, (for example, if you are using Click&Scout), this function will take the starting zone and tempo of the attack to map it to an attack code.

Usage

```
dv_attack_code_map(type, start_zone)
```

Arguments

`type`	character: vector of attack tempos ("H", "T", "Q", etc). A `type` vector of length 1 will be expanded to the length of the `start_zone` vector, if needed
`start_zone`	integer: vector of start zones

Value

A vector of attack codes, set_types, etc.

Examples

```
dv_attack_code_map(type = c("H", "Q", "T"), start_zone = c("8", "3", "4"))
```

`dv_attack_phase` *Attack phase*

Description

Attack phase as defined by DataVolley: either "Reception", "Transition sideout" or "Transition breakpoint", assigned only to attack actions.

Usage

```
dv_attack_phase(x)
```

Arguments

`x`	datavolleyplays: the plays component of a datavolley object as returned by [dv_read()]

Value

Character vector

dv_cone2xy *Attack cones to x, y coordinates*

Description

Attack cones to x, y coordinates

Usage

```
dv_cone2xy(
  start_zones,
  end_cones,
  end = "upper",
  xynames = c("ex", "ey"),
  as = "points",
  force_center_zone = FALSE
)
```

Arguments

start_zones	integer: starting zone of attack
end_cones	integer: cone of attack
end	string: use the "lower" or "upper" part of the figure
xynames	character: names to use for the x and y columns of the returned data.frame
as	string: either "points" or "polygons" (see Value, below)
force_center_zone	logical: a vector indicating the attacks that should be treated as center zone attacks regardless of their start_zone value (e.g. by the setter). If FALSE, the start_zone value will be used. If provided as a single scalar value, this will be applied to all attacks

Value

a tibble (NOT a data.frame) with columns "x" and "y" (or other names if specified in xynames). If as is "polygons", the columns will be lists, because each polygon will have four x- and y-coordinates

See Also

ggcourt, dv_flip_xy, dv_xy2index, dv_index2xy, dv_xy, dv_xy2zone, dv_xy2subzone

Examples

```
## Not run:
## attacks from left side (zone 4) to cones 1-7

## plot as line segments
```

```
cxy <- dv_cone2xy(4, 1:7)
## add starting coordinate for zone 4
cxy <- cbind(dv_xy(4), cxy)
ggplot(cxy, aes(x, y, xend=ex, yend=ey)) + geom_segment() + ggcourt()

## plot as polygons
cxy <- dv_cone2xy(4, 1:7, as = "polygons")

## this returns coordinates as list columns, unpack these to use with ggplot
##  also add an identifier for each polygon
cxy <- data.frame(x = unlist(cxy$ex), y = unlist(cxy$ey),
                  id = unlist(lapply(1:nrow(cxy), rep, 4)))
ggplot(cxy, aes(x, y, group = id, fill = as.factor(id))) + geom_polygon() +
   ggcourt()

## End(Not run)
```

dv_cone_polygons *The polygon coordinates for attack cones*

Description

The polygon coordinates for attack cones

Usage

```
dv_cone_polygons(zone, end = "upper", extended = FALSE)
```

Arguments

zone	string: one of "L", "R", "M"
end	string: use the "lower" or "upper" part of the figure
extended	logical: if FALSE, the polygons will only cover the actual court area; if TRUE, they will be extended to cover the court periphery as well

Value

A data.frame with columns cone_number, x, y

Examples

```
## Not run:
 library(ggplot2)
 cxy <- dv_cone_polygons("M")
 ggplot(cxy, aes(x, y, group = cone_number, fill = as.factor(cone_number))) +
   geom_polygon() + ggcourt()

## End(Not run)
```

dv_court *Plot a volleyball court diagram*

Description

Volleyball court schematic suitable for adding to a figure

Usage

```
dv_court(
  plot_package = "base",
  court = "full",
  show_zones = TRUE,
  labels = c("Serving team", "Receiving team"),
  as_for_serve = FALSE,
  show_zone_lines = TRUE,
  show_minor_zones = FALSE,
  grid_colour = "black",
  zone_colour = "grey70",
  minor_zone_colour = "grey80",
  fixed_aspect_ratio = TRUE,
  zone_font_size = 10,
  ...
)
```

Arguments

`plot_package` string: either "base" or "ggplot2". If "ggplot2", the `ggcourt` function is used

`court` string: "full" (show full court) or "lower" or "upper" (show only the lower or upper half of the court)

`show_zones` logical: add numbers indicating the court zones (3m squares)?

`labels` string: labels for the lower and upper court halves (pass NULL for no labels)

`as_for_serve` logical: if TRUE and `show_zones` is TRUE, show zones as for serving. Only zones 1,5,6,7,9 are meaningful in this case

`show_zone_lines`
logical: if FALSE, just show the 3m line. If TRUE, also show the 3m x 3m zones

`show_minor_zones`
logical: add lines for the subzones (1.5m squares)?

`grid_colour` string: colour to use for court sidelines, 3m line, and net

`zone_colour` string: colour to use for zone lines and labels

`minor_zone_colour`
string: colour to use for minor zone grid lines

`fixed_aspect_ratio`
logical: if TRUE, coerce the plotted court to be square (for a half-court plot) or a 2:1 rectangle (full court plot). Prior to package version 0.5.3 this was not TRUE by default

`zone_font_size` numeric: the font size of the zone labels

`...` : additional parameters passed to `ggplot2::theme_classic(...)`

Details

The datavolley package uses the following dimensions and coordinates for plotting:

- the court is shown such that the sidelines are oriented vertically and the net is oriented horizontally
- the intersection of the left-hand sideline and the bottom baseline is at (0.5, 0.5)
- the intersection of the right-hand sideline and the top baseline is at (3.5, 6.5)
- the net intersects the sidelines at (0.5, 3.5) and (3.5, 3.5)
- the zones 1-9 (as defined in the DataVolley manual) on the lower half of the court are located at:
  1. (3, 1)
  2. (3, 3)
  3. (2, 3)
  4. (1, 3)
  5. (1, 1)
  6. (2, 1)
  7. (1, 2)
  8. (2, 2)
  9. (3, 2)
- the zones 1-9 (as defined in the DataVolley manual) on the upper half of the court are located at:
  1. (1, 6)
  2. (1, 4)
  3. (2, 4)
  4. (3, 4)
  5. (3, 6)
  6. (2, 6)
  7. (3, 5)
  8. (2, 5)
  9. (1, 5)

To get a visual depiction of this, try: `ggplot() + ggcourt() + theme_bw()`

See Also

`ggcourt` for a `ggplot2` equivalent function; `dv_xy`, `dv_xy2index`, `dv_index2xy`, `dv_flip_xy`

Examples

```
## Not run:
x <- dv_read(dv_example_file(), insert_technical_timeouts=FALSE)

library(dplyr)

## Example: attack frequency by zone, per team

attack_rate <- plays(x) %>% dplyr::filter(skill == "Attack") %>%
  group_by(team, start_zone) %>% dplyr::summarize(n_attacks = n()) %>%
  mutate(rate = n_attacks/sum(n_attacks)) %>% ungroup

## add columns "x" and "y" for the x,y coordinates associated with the zones
attack_rate <- cbind(attack_rate, dv_xy(attack_rate$start_zone, end = "lower"))

## for team 2, these need to be on the top half of the diagram
tm2 <- attack_rate$team == teams(x)[2]
attack_rate[tm2, c("x", "y")] <- dv_xy(attack_rate$start_zone, end = "upper")[tm2, ]

## plot it
dv_heatmap(attack_rate[, c("x", "y", "rate")], legend_title = "Attack rate")

## add the court diagram
dv_court(labels = teams(x))

## End(Not run)
```

dv_create_meta_attacks

Create a meta attack data.frame from the plays object if it is missing

Description

If your DataVolley file does not have a meta attack dataframe (for example, if you are using Click&Scout), this function will create one from the information in the plays object.

Usage

```
dv_create_meta_attacks(plays)
```

Arguments

`plays`	data.frame: the plays component of a datavolley object, as returned by `dv_read`

Value

A data.frame of attacks.

dv_example_file *Example DataVolley files provided as part of the datavolley package*

Description

Example DataVolley files provided as part of the datavolley package

Usage

```
dv_example_file(choice = 1)
```

Arguments

choice numeric: which data file to return?

- 1 - the January 2015 Slovenian junior women's final between Braslovče and Nova KBM Branik (obtained from http://www.odbojka.si/
- 2 - the December 2012 men's Slovenian national championship semifinal between ACH Volley and Maribor (obtained from http://www.odbojka.si/
- 3 - Nicaragua vs Cuba women from the Pan American Cup, August 2022 (vsm format, courtesy Christophe Elek)

Value

path to the file

See Also

dv_read

Examples

```
myfile <- dv_example_file()
x <- dv_read(myfile, insert_technical_timeouts = FALSE)
summary(x)
```

dv_fake_coordinates *Fake coordinate data*

Description

Generates fake coordinate data. The DataVolley software has the capability to accurately record court locations associated with each action. However, not all files contain this information (it can be time consuming to enter). This function generates fake coordinate data that can be used for demonstration purposes.

Usage

```
dv_fake_coordinates(skill, evaluation)
```

Arguments

skill	string: the skill type to generate positions for (only "serve" is implemented so far)
evaluation	character: vector of evaluations (as returned in the evalution column of a datavolleyplays object)

Value

data.frame of coordinates with columns "start_coordinate", "start_coordinate_x", "start_coordinate_y", "end_coordinate", "end_coordinate_x", "end_coordinate_y". The returned data.frame will have as many rows as the length of the evaluation vector

See Also

dv_xy

Examples

```
## Not run:
library(ggplot2)

## read example data file
x <- dv_read(dv_example_file(), insert_technical_timeouts = FALSE)

## take just the serves from the play-by-play data
xserves <- subset(plays(x), skill=="Serve")

## if the file had been scouted with coordinate included, we could plot them directly
## this file has no coordinates, so we'll fake some up for demo purposes
coords <- dv_fake_coordinates("serve", xserves$evaluation)
xserves[, c("start_coordinate", "start_coordinate_x", "start_coordinate_y",
            "end_coordinate", "end_coordinate_x", "end_coordinate_y")] <- coords

## now we can plot these
```

```
xserves$evaluation[!xserves$evaluation %in% c("Ace", "Error")] <- "Other"

ggplot(xserves, aes(start_coordinate_x, start_coordinate_y,
       xend=end_coordinate_x, yend=end_coordinate_y, colour=evaluation))+
    geom_segment() + geom_point() +
  scale_colour_manual(values=c(Ace="limegreen", Error="firebrick", Other="dodgerblue")) +
    ggcourt(labels=c("Serving team", "Receiving team"))

## End(Not run)
```

dv_find_to_flip_coordinates

Find coordinates that need flipping

Description

The orientation of coordinates (e.g. is a serve going from the lower part of the court to the upper, or vice-versa?) depends on how the scout entered them. This function finds coordinates that require flipping, so that all attacks/serves/whatever can be plotted with the same orientation

Usage

```
dv_find_to_flip_coordinates(x, target_start_end = "lower")
```

Arguments

x	datavolleyplays: the plays component of a datavolley object as returned by dv_read
target_start_end	string: "lower" or "upper"

Value

A logical index with length equal to the number of rows of x. TRUE indicates rows of x that need their coordinates flipped

See Also

dv_flip_xy

dv_flip_xy *Flip the x,y court coordinates*

Description

This is a convenience function that will transform coordinates from the top half of the court to the bottom, or vice-verse.

Usage

```
dv_flip_xy(x, y)

dv_flip_x(x)

dv_flip_y(y)

dv_flip_index(index)
```

Arguments

x	numeric: x-coordinate. For dv_flip_xy this can be a two-column matrix or data.frame containing x and y
y	numeric: y-coordinate
index	integer: grid index value

Value

transformed coordinates or grid index

See Also

ggcourt, dv_xy, dv_xy2index, dv_index2xy

Examples

```
## Not run:
 x <- dv_read(dv_example_file(), insert_technical_timeouts=FALSE)
 library(ggplot2)
 library(dplyr)

## attack rate by zone (both teams combined)
attack_rate <- plays(x) %>% dplyr::filter(skill=="Attack") %>%
   group_by(team, start_zone) %>% dplyr::summarize(n_attacks=n()) %>%
   mutate(rate=n_attacks/sum(n_attacks)) %>% ungroup

## add columns "x" and "y" for the x,y coordinates associated with the zones
attack_rate <- cbind(attack_rate, dv_xy(attack_rate$start_zone, end="lower"))
```

```
## plot this
ggplot(attack_rate, aes(x, y, fill=rate)) + geom_tile() + ggcourt(labels=teams(x)) +
     scale_fill_gradient2(name="Attack rate")

## or, plot at the other end of the court
attack_rate <- attack_rate %>% mutate(x=dv_flip_x(x), y=dv_flip_y(y))

ggplot(attack_rate, aes(x, y, fill=rate)) + geom_tile() + ggcourt(labels=teams(x)) +
     scale_fill_gradient2(name="Attack rate")

## End(Not run)
```

dv_heatmap *Plot a court heatmap, using base graphics*

Description

See `link{ggcourt}` for a `ggplot2`-based court diagram, which can be used to plot heatmaps with e.g. `ggplot2::geom_tile`.

Usage

```
dv_heatmap(
  x,
  y,
  z,
  col,
  zlim,
  legend = TRUE,
  legend_title = NULL,
  legend_title_font = 1,
  legend_title_cex = 0.7,
  legend_cex = 0.7,
  legend_pos = c(0.8, 0.85, 0.25, 0.75),
  res,
  add = FALSE
)
```

Arguments

`x`	numeric, RasterLayer or data.frame: x-coordinates of the data to plot, or a `RasterLayer` layer or `data.frame` containing the data (x, y, and z together)
`y`	numeric: y-coordinates of the data to plot
`z`	numeric: values of the data to plot
`col`	character: a vector of colours to use

`zlim` numeric: the minimum and maximum z values for which colors should be plotted, defaulting to the range of the finite values of z

`legend` logical: if `TRUE`, plot a legend

`legend_title` string: title for the legend

`legend_title_font`
numeric: 1 = normal, 2 = bold, 3 = italic

`legend_title_cex`
numeric: size scaling of legend title

`legend_cex` numeric: size scaling of legend text

`legend_pos` numeric: position of the legend (xmin, xmax, ymin, ymax) - in normalized units

`res` numeric: size of the heatmap cells. This parameter should only be needed in cases where the input data are sparse, when the automatic algorithm can't work it out. Values are given in metres, so `res` is 3 when showing zones, or 1.5 when showing subzones

`add` logical: if `TRUE`, add the heatmap to an existing plot

Details

Data can be provided either as separate `x`, `y`, and `z` objects, or as a single `RasterLayer` or `data.frame` object. If a `data.frame`, the first three columns are used (and assumed to be in the order `x`, `y`, `z`).

See Also

`dv_court`, `dv_plot_new`

Examples

```
## Not run:
x <- dv_read(dv_example_file(), insert_technical_timeouts = FALSE)

library(dplyr)

## Example: attack frequency by zone, per team

attack_rate <- plays(x) %>% dplyr::filter(skill == "Attack") %>%
  group_by(team, start_zone) %>% dplyr::summarize(n_attacks = n()) %>%
  mutate(rate = n_attacks/sum(n_attacks)) %>% ungroup

## add columns "x" and "y" for the x,y coordinates associated with the zones
attack_rate <- cbind(attack_rate, dv_xy(attack_rate$start_zone, end = "lower"))

## for team 2, these need to be on the top half of the diagram
tm2 <- attack_rate$team == teams(x)[2]
attack_rate[tm2, c("x", "y")] <- dv_xy(attack_rate$start_zone, end="upper")[tm2, ]

## plot it
dv_heatmap(attack_rate[, c("x", "y", "rate")], legend_title = "Attack rate")

## or, controlling the z-limits
```

```
dv_heatmap(attack_rate[, c("x", "y", "rate")], legend_title = "Attack rate", zlim = c(0, 1))

## add the court diagram
dv_court(labels = teams(x))

## sometimes you may need more control over the plot layout
## set up a plot with 10% bottom/top margins and 20% left/right margins
## showing the lower half of the court only
dv_plot_new(margins = c(0.05, 0.1, 0.05, 0.1), court = "lower")
## add the heatmap
dv_heatmap(attack_rate[1:6, c("x", "y", "rate")], add = TRUE)
## and the court diagram
dv_court(court = "lower")


## End(Not run)
```

dv_index2xy — *Grid index to x,y coordinate and vice-versa*

Description

DataVolley uses a grid to represent positions on court (values in columns "start_coordinate", "mid_coordinate", and "end_coordinate" in the play-by-play data frame). These functions convert grid index values to x, y coordinates suitable for plotting, and vice-versa. For a description of the court dimensons and coordinates see ggcourt.

Usage

```
dv_index2xy(index)

dv_xy2index(x, y)
```

Arguments

index	integer: vector of grid indices. If missing, the entire grid will be returned. The row numbers match the grid indices
x	numeric: x-coordinate. For dv_index2xy this can be a two-column matrix or data.frame containing x and y
y	numeric: y-coordinate

Value

for dv_index2xy, a data.frame with columns "x" and "y"; for dv_xy2index a vector of integer values

See Also

ggcourt, dv_xy, dv_flip_xy, dv_xy2zone, dv_xy2subzone

Examples

```
## positions (zones) 1 and 3 are at x, y coordinates c(3, 1) and c(2, 3) respectively

## their grid indices:
dv_xy2index(c(3, 2), c(1, 3))
```

dv_int2rgb *Convert integer colour to RGB*

Description

DataVolley files use an integer representation of colours. These functions convert to and from hex colour strings as used in R.

Usage

```
dv_int2rgb(z)

dv_rgb2int(x)
```

Arguments

z	integer: vector of integers
x	integer: vector of hex colour strings

Value

Character vector of hex RGB colour strings

Examples

```
dv_int2rgb(c(255, 16711680))
```

dv_meta_video *Get or set the video metadata in a datavolley object*

Description

Get or set the video metadata in a datavolley object

Usage

```
dv_meta_video(x)

dv_meta_video(x) <- value
```

Arguments

x	datavolley: a datavolley object as returned by [datavolley::dv_read()]
value	string or data.frame: a string containing the path to the video file, or a data.frame with columns "camera" and "file"

Value

For ‘dv_meta_video‘, the existing video metadata. For ‘dv_meta_video<-‘, the video metadata value in ‘x‘ is changed

Examples

```
x <- dv_read(dv_example_file())
dv_meta_video(x) ## empty dataframe
dv_meta_video(x) <- "/path/to/my/videofile"
dv_meta_video(x)
```

dv_plot_new *Create a new plot page for base graphics plotting*

Description

The plot will be set up as either a full- or half-court plot, depending on the inputs. The extent can be specified via the court argument (values either "full", "lower", or "upper"), or via the x and y arguments. If the latter, provide either separate x and y numeric vectors, or as a single x RasterLayer object. If no extent is specified by any of these methods, a full-court plot is assumed.

Usage

```
dv_plot_new(x, y, legend, court, margins, par_args, ...)
```

Arguments

x	numeric or RasterLayer: x-coordinates of the data to plot, or a RasterLayer layer defining the extent of the data
y	numeric: y-coordinates of the data to plot. Not needed if x is a RasterLayer object
legend	logical: if TRUE, leave space for a legend
court	string: either "full", "lower", or "upper"
margins	numeric: vector of four values to use as margins (bottom, left, top, right). Values are as a proportion of the plot size
par_args	list: parameters to pass to par
...	: additional parameters passed to plot.window

See Also

dv_court, dv_heatmap

Examples

```
dv_plot_new()
## show an attack from position 4 to position 6
from <- dv_xy(4, end = "lower")
to <- dv_xy(6, end = "upper")
lines(c(from[1], to[1]), c(from[2], to[2]), col = "green")
## add the court diagram
dv_court(labels = c("Attacking team", "Defending team"))
```

---

dv_point_phase *Point phase*

Description

Point phase as defined by DataVolley: either "Sideout" or "Breakpoint", assigned only to winning or losing actions (including green codes). Note that the point phase is inferred for the winning action (i.e. the point phase value for both the winning and losing action is "Sideout" if the winning team was receiving).

Usage

```
dv_point_phase(x)
```

Arguments

x	datavolleyplays: the plays component of a datavolley object as returned by [dv_read()]

Value

Character vector

dv_read *Read a datavolley file*

Description

The do_transliterate option may be helpful when trying to work with multiple files from the same competition, since different text encodings may be used on different files. This can lead to e.g. multiple versions of the same team name. Transliterating can help avoid this, at the cost of losing e.g. diacriticals. Transliteration is applied after converting from the specified text encoding to UTF-8. Common encodings used with DataVolley files include "windows-1252" (western Europe), "windows-1250" (central Europe), "iso-8859-1" (western Europe and Americas), "iso-8859-2" (central/eastern Europe), "iso-8859-13" (Baltic languages)

Usage

```
dv_read(
  filename,
  insert_technical_timeouts = TRUE,
  do_warn = FALSE,
  do_transliterate = FALSE,
  encoding = "guess",
  date_format = "guess",
  extra_validation = 2,
  validation_options = list(),
  surname_case = "asis",
  skill_evaluation_decode = "default",
  custom_code_parser,
  metadata_only = FALSE,
  verbose = FALSE,
  edited_meta
)

read_dv(
  filename,
  insert_technical_timeouts = TRUE,
  do_warn = FALSE,
  do_transliterate = FALSE,
  encoding = "guess",
  date_format = "guess",
  extra_validation = 2,
  validation_options = list(),
  surname_case = "asis",
  skill_evaluation_decode = "default",
  custom_code_parser,
  metadata_only = FALSE,
  verbose = FALSE,
  edited_meta
)
```

Arguments

`filename` string: file name to read

`insert_technical_timeouts`
logical or list: should we insert technical timeouts? If TRUE, technical timeouts are inserted at points 8 and 16 of sets 1–4 (for indoor files) or when the team scores sum to 21 in sets 1–2 (beach). Otherwise a two-element list can be supplied, giving the scores at which technical timeouts will be inserted for sets 1–4, and set 5.

`do_warn` logical: should we issue warnings about the contents of the file as we read it?

`do_transliterate`
logical: should we transliterate all text to ASCII? See details

`encoding` character: text encoding to use. Text is converted from this encoding to UTF-8. A vector of multiple encodings can be provided, and this function will attempt to choose the best. If encoding is "guess", the encoding will be guessed

`date_format` string: the expected date format (one of "ymd", "mdy", or "dmy") or "guess". If `date_format` is something other than "guess", that date format will be preferred where dates are ambiguous

`extra_validation`
numeric: should we run some extra validation checks on the file? 0=no extra validation, 1=check only for major errors, 2=somewhat more extensive, 3=the most extra checking

`validation_options`
list: additional options to pass to the validation step. See `dv_validate` for details

`surname_case` string or function: should we change the case of player surnames? If `surname_case` is a string, valid values are "upper","lower","title", or "asis"; otherwise `surname_case` may be a function that will be applied to the player surname strings

`skill_evaluation_decode`
function or string: if `skill_evaluation_decode` is a string, it can be either "default" (use the default DataVolley conventions for dvw or vsm files), "volleymetrics" (to follow the scouting conventions used by VolleyMetrics), "german" (same as "default" but with B/ and B= swapped), or "guess" (use volleymetrics if it looks like a VolleyMetrics file, otherwise default). If `skill_evaluation_decode` is a function, it should convert skill evaluation codes into meaningful phrases. See `skill_evaluation_decoder`

`custom_code_parser`
function: function to process any custom codes that might be present in the datavolley file. This function takes one input (the `datavolley` object) and should return a list with two named components: `plays` and `messages`

`metadata_only` logical: don't process the plays component of the file, just the match and player metadata

`verbose` logical: if TRUE, show progress `edited_meta` list: [very much experimental] if supplied, will be used in place of the metadata present in the file itself. This makes it possible to, for example, read a file, edit the metadata, and re-parse the file but using the modified metadata

Value

A named list with several elements. `meta` provides match metadata, `plays` is the main play-by-play data in the form of a data.frame. `raw` is the line-by-line content of the datavolley file. `messages` is a data.frame describing any inconsistencies found in the file.

References

http://www.dataproject.com/IT/en/Volleyball

See Also

skill_evaluation_decoder dv_validate

Examples

```
## Not run:
  ## to read the example file bundled with the package
  myfile <- dv_example_file()
  x <- dv_read(myfile, insert_technical_timeouts=FALSE)
  summary(x)

  ## or to read your own file:
  x <- dv_read("c:/some/path/myfile.dvw", insert_technical_timeouts=FALSE)

  ## Insert a technical timeout at point 12 in sets 1 to 4:
  x <- dv_read(myfile, insert_technical_timeouts=list(c(12),NULL))

  ## to read a VolleyMetrics file
  x <- dv_read(myfile, skill_evaluation_decode = "volleymetrics")

## End(Not run)
```

dv_read_sq *Read a team roster (*.sq) file*

Description

Read a team roster (*.sq) file

Usage

```
dv_read_sq(
  filename,
  do_transliterate = FALSE,
  encoding = "guess",
  date_format = "guess",
  surname_case = "asis",
  verbose = FALSE
)
```

Arguments

`filename` string: file name to read

`do_transliterate`
logical: should we transliterate all text to ASCII?

`encoding` character: text encoding to use. Text is converted from this encoding to UTF-8. A vector of multiple encodings can be provided, and this function will attempt to choose the best. If encoding is "guess", the encoding will be guessed

`date_format` string: the expected date format (used for dates of birth). One of "ymd", "mdy", "dmy", or "guess". If `date_format` is something other than "guess", that date format will be preferred where dates are ambiguous

`surname_case` string or function: should we change the case of player surnames? If `surname_case` is a string, valid values are "upper","lower","title", or "asis"; otherwise `surname_case` may be a function that will be applied to the player surname strings

`verbose` logical: if `TRUE`, show progress

Value

A list with two components: "team" and "players", both of which are data frames

Examples

```
## Not run:
  x <- dv_read_sq("/path/to/my/roster_file")

## End(Not run)
```

dv_repair *Attempt to repair a datavolley object*

Description

Currently an attempt will be made to repair these issues: * if multiple players on the same team have the same jersey number, players with that number (on that team) who did not take to the court will be removed from their team roster. In this situation, whether or not a player took to the court is determined from the match metadata only * if multiple players have the same player ID but different jersey numbers, players with that ID who did not take to the court will be removed from their team roster. In this situation, whether or not a player took to the court is determined from the match metadata and the play-by-play data

Usage

```
dv_repair(x)
```

Arguments

x	datavolley: a datavolley object as returned by [dv_read()]

Value

A modified copy of ‘x‘. If problems exist and cannot be repaired, an error will be thrown

---

dv_sync_summary *Summarize the video sync times in a dvw file*

Description

This function will generate a summary of various video time differences in a dvw file. Apply this to a file that you have synchronized to video, and the results can be used to tweak the behaviour of `dv_sync_video`.

Usage

```
dv_sync_summary(x)
```

Arguments

x	datavolley: a single datavolley object as returned by `dv_read`, or the `plays` component of one

Value

A data.frame with columns `type`, `N`, `mean`, `most_common`, `min`, `max`

See Also

`dv_sync_video`

Examples

```
x <- dv_read(dv_example_file(3))
dv_sync_summary(x)
```

dv_sync_video *Synchronize video times*

Description

This function uses the time of each serve and some rules to align the other contacts in a rally with their (approximately correct) times in the corresponding match video. Warning: experimental!

Usage

```
dv_sync_video(
  x,
  first_serve_contact,
  freeball_dig_time_offset = NA,
  contact_times = dv_sync_contact_times(),
  offsets = dv_sync_offsets(),
  times_from,
  enforce_order = TRUE
)

dv_sync_contact_times(...)

dv_sync_offsets(...)
```

Arguments

`x` datavolley: a single datavolley object as returned by `dv_read`

`first_serve_contact`
numeric or string: the video time of the first serve contact. This can be a numeric value giving the time in seconds from the start of the video, or a string of the form "MM:SS" (minutes and seconds) or "HH:MM:SS" (hours, minutes and seconds)

`freeball_dig_time_offset`
numeric: if non-NA, the clock times of freeball digs will be used directly in the synchronization process. Freeball digs will be aligned using their clock times relative to the first serve contact clock time, with this `freeball_dig_time_offset` value (in seconds) added. So if when scouting live you typically enter freeball digs one second after they happen, use `freeball_dig_time_offset = -1`. If `freeball_dig_time_offset` is NA, which is the default, the clock times of freeball digs will not be used in the synchronization process

`contact_times` list: a set of parameters that control the synchronization process. See Details, below `offsets` list: a list set of offsets to be added to each contact time in the second step of the synchronization process. See Details, below. If `offsets` is NULL or an empty list, no offsets are applied

`times_from` string: either "clock" or "video": take the serve times (and freeball dig times, if `freeball_dig_time_offset` is non-NA) from clock or video times. By default, clock times are used unless they are all missing

`enforce_order` logical: the estimated contact times will always be time-ordered (the contact time of a given touch cannot be prior to the contact time of a preceding touch). But the offsets can be different for different skills, leading to final video times that are not time ordered. These will be fixed if `enforce_order` is TRUE

`...` : name-value pairs of elements to override the defaults in `dv_sync_contact_times` and `dv_sync_offsets`

Details

When a match is scouted live, the clock time of each serve will usually be correct because the scout can enter the serve code at the actual time of serve. But the remainder of the touches in the rally might not be at their correct times if the scout can't keep up with the live action. This function makes some assumptions about typical contact-to-contact times to better synchronize the scouted contacts with the corresponding match video.

The clock time of each serve will be used as the reference time for each rally (unless the user specifies `times_from = "video"`). If clock times are not present in the file, the video time of each serve will be used instead. If those are also missing, the function will fail.

Freeball digs can optionally be treated in the same way as serves, with their scouted times used directly in the synchronization process. Obviously this only makes sense if the scout has actually been consistent in their timing when entering freeball digs, but assuming that is the case then setting the `freeball_dig_time_offset` to a non-NA value will improve the synchronization of rallies with freeballs. These rallies otherwise tend to synchronize poorly, because the play is messy and less predictable compared to in-system rallies.

Note that synchronization from clock times relies on the serve clock times in the file being consistent, and so it will only work if the match has been scouted in a single sitting (either live, or from video playback but without pausing/rewinding/fast-forwarding the video). If your clock times are not consistent but the video time of each serve is correct, then you can use the video time of each serve as the reference time instead.

The synchronization is a two-step process. In the first step, the video time of each scouted contact is estimated (i.e. the actual time that the player made contact with the ball). In the second step, skill-specific offsets are added to those contact times. (This is important if your video montage software uses the synchronized video times directly, because you will normally want a video clip to start some seconds before the actual contact of interest).

The `contact_times` object contains a set of times (in seconds), which you can adjust to suit your scouting style and level of play. If you have an already-synchronized dvw file, the `dv_sync_summary` function can provide some guidance as to what these values should be. The `contact_times` object contains the following entries:

- SQ - time between the scouted serve time and actual serve contact for jump serves
- SM - time between the scouted serve time and actual serve contact for jump-float serves

- SO - time between the scouted serve time and actual serve contact for all other serves
- SQ_R, SM_R, SO_R - the time between serve contact and reception contact for jump, jump-float, and other serves
- R_E - the time between reception contact and set contact
- EQ_A - the time between set contact and attack contact for quick sets
- EH_A - the time between set contact and attack contact for high sets
- EO_A - the time between set contact and attack contact for all other sets
- A_B - the time between attack contact and block contact
- A_D - the time between attack contact and dig contact (no intervening block touch)
- A_B_D - the time between attack contact and dig contact (with block touch)
- D_E - the time between dig contact and set contact
- RDov - the time between reception or dig overpass contact and the next touch by the opposition
- END - the time between the last contact and end-of-rally marker

The `offsets` object defines the offset (in seconds) to be added to each contact time in the second pass of the synchronization process. It contains the entries "S" (serve), "R" (reception), "E" (set), "A" (attack), "D", (dig), "B" (block), and "F" (freeball).

Note that the entries in `contact_times` and `offsets` can be fractions. The actual video time entries in the returned file are required to be integers and so the final values will be rounded, but using fractional values (particularly for the `contact_times` entries) can give better accuracy in the intermediate calculations.

Value

A copy of `x` with modified `video_time` values in its `plays` component

See Also

`dv_sync_summary`

Examples

```
x <- dv_read(dv_example_file())
## first serve contact was at 54s in the video
x <- dv_sync_video(x, first_serve_contact = 54)

## with a custom configuration
my_contact_times <- dv_sync_contact_times(SQ = 3) ## override default entries as necessary
## first serve contact was at 3:35 in the video
x <- dv_sync_video(x, first_serve_contact = "3:35", contact_times = my_contact_times)
```

`dv_validate` *Additional validation checks on a DataVolley file*

Description

This function is automatically run as part of `dv_read` if `extra_validation` is greater than zero. The current validation messages/checks are:

- message "The total of the [home|visiting] team scores in the match result summary (x$meta$result) does not match the total number of points recorded for the [home|visiting] team in the plays data"
- message "[Home|Visiting] team roster is empty": the home or visiting team roster has not been entered
- message "Players xxx and yyy have the same player ID": player IDs should be unique, and so duplicated IDs will be flagged here
- message "Players xxx and yyy have the same jersey number": players on the same team should not have the same jersey number
- message "The listed player is not on court in this rotation": the player making the action is not part of the current rotation. Libero players are ignored for this check
- message "Back-row player made an attack from a front-row zone": an attack starting from zones 2-4 was made by a player in the back row of the current rotation
- message "Front-row player made an attack from a back-row zone (legal, but possibly a scouting error)": an attack starting from zones 1,5-9 was made by a player in the front row of the current rotation
- message "Quick attack by non-middle player"
- message "Middle player made a non-quick attack"
- message "Block by a back-row player"
- message "Winning serve not coded as an ace"
- message "Non-winning serve was coded as an ace"
- message "Serving player not in position 1"
- message "Player designated as libero was recorded making a [serve|attack|block]"
- message "Attack (which was blocked) does not have number of blockers recorded"
- message "Attack (which was followed by a block) has 'No block' recorded for number of players"
- message "Repeated row with same skill and evaluation_code for the same player"
- message "Consecutive actions by the same player"
- message "Point awarded to incorrect team following error (or \"error\" evaluation incorrect)"
- message "Point awarded to incorrect team (or [winning play] evaluation incorrect)"
- message "Scores do not follow proper sequence": one or both team scores change by more than one point at a time

- message "Visiting/Home team rotation has changed incorrectly"
- message "Player lineup did not change after substitution: was the sub recorded incorrectly?"
- message "Player lineup conflicts with recorded substitution: was the sub recorded incorrectly?"
- message "Reception type does not match serve type": the type of reception (e.g. "Jump-float serve reception" does not match the serve type (e.g. "Jump-float serve")
- message "Reception start zone does not match serve start zone"
- message "Reception end zone does not match serve end zone"
- message "Reception end sub-zone does not match serve end sub-zone"
- message "Attack type ([type]) does not match set type ([type])": the type of attack (e.g. "Head ball attack") does not match the set type (e.g. "High ball set")
- message "Block type ([type]) does not match attack type ([type])": the type of block (e.g. "Head ball block") does not match the attack type (e.g. "High ball attack")
- message "Dig type ([type]) does not match attack type ([type])": the type of dig (e.g. "Head ball dig") does not match the attack type (e.g. "High ball attack")
- message "Multiple serves in a single rally"
- message "Multiple receptions in a single rally"
- message "Serve (that was not an error) did not have an accompanying reception"
- message "Rally had ball contacts but no serve"

Usage

```
dv_validate(x, validation_level = 2, options = list(), file_type)

validate_dv(x, validation_level = 2, options = list(), file_type)
```

Arguments

`x` datavolley: datavolley object as returned by `dv_read`

`validation_level`
numeric: how strictly to check? If 0, perform no checking; if 1, only identify major errors; if 2, also return any issues that are likely to lead to misinterpretation of data; if 3, return all issues (including minor issues such as those that might have resulted from selective post-processing of compound codes)

`options` list: named list of options that control optional validation behaviour. Valid entries are:

- setter_tip_codes character: vector of attack codes that represent setter tips (or other attacks that a back-row player can validly make from a front-row position). If you code setter tips as attacks, and don't want such attacks to be flagged as an error when made by a back-row player in a front-row zone, enter the setter tip attack codes here. e.g. `options=list(setter_tip_codes=c("PP","XY"))`

`file_type` string: "indoor" or "beach". If not provided, will be taken from the `x$file_meta$file_format` entry

Value

data.frame with columns message (the validation message), file_line_number (the corresponding line number in the DataVolley file), video_time, and file_line (the actual line from the DataVolley file).

See Also

`dv_read`

Examples

```
## Not run:
  x <- dv_read(dv_example_file(), insert_technical_timeouts = FALSE)
  xv <- dv_validate(x)

  ## specifying "PP" as the setter tip code
  ## front-row attacks (using this code) by a back-row player won't be flagged as errors
  xv <- dv_validate(x, options = list(setter_tip_codes = c("PP")))

## End(Not run)
```

`dv_write` *Write a datavolley object to dvw file*

Description

Note that this is really rather experimental, and you probably shouldn't use it yet. Once complete, this function will allow a datavolley file to be read in via `dv_read`, modified by the user, and then rewritten back to a datavolley file. At this stage, most modifications to the datavolley object should make it back into the rewritten file. However, the scouted code (in the `code` column) is NOT yet updated to reflect changes that might have been made to other columns in the datavolley object.

Usage

```
dv_write(x, file, text_encoding = "UTF-8")

write_dv(x, file, text_encoding = "UTF-8")
```

Arguments

`x` datavolley: a datavolley object as returned by `dv_read`

`file` string: the filename to write to. If not supplied, no file will be written but the dvw content will be returned

`text_encoding` string: the text encoding to use

Value

The dvw file contents as a character vector (invisibly)

See Also

dv_read

Examples

```
## Not run:
  x <- dv_read(dv_example_file())
  outfile <- tempfile()
  dv_write(x, outfile)

## End(Not run)
```

dv_xy   *Court zones to x, y coordinates*

Description

Generate x and y coordinates for plotting, from DataVolley numbered zones

Usage

```
dv_xy(
  zones,
  end = "lower",
  xynames = c("x", "y"),
  as_for_serve = FALSE,
  subzones
)
```

Arguments

zones	numeric: zones numbers 1-9 to convert to x and y coordinates
end	string: use the "lower" or "upper" part of the figure
xynames	character: names to use for the x and y columns of the returned data.frame
as_for_serve	logical: if TRUE, treat positions as for serving. Only zones 1,5,6,7,9 are meaningful in this case
subzones	character: if supplied, coordinates will be adjusted for subzones. Values other than "A" to "D" will be ignored

Details

For a description of the court dimensions and coordinates used for plotting, see ggcourt

Value

data.frame with columns "x" and "y" (or other names if specified in xynames)

See Also

ggcourt, dv_flip_xy, dv_xy2index, dv_index2xy, dv_cone2xy, dv_xy2zone, dv_xy2subzone

Examples

```
## Not run:
x <- dv_read(dv_example_file(), insert_technical_timeouts = FALSE)

library(ggplot2)
library(dplyr)

## Example 1: attack frequency by zone, per team

attack_rate <- plays(x) %>% dplyr::filter(skill == "Attack") %>%
  group_by(team, start_zone) %>% dplyr::summarize(n_attacks = n()) %>%
  mutate(rate = n_attacks/sum(n_attacks)) %>% ungroup

## add columns "x" and "y" for the x, y coordinates associated with the zones
attack_rate <- cbind(attack_rate, dv_xy(attack_rate$start_zone, end = "lower"))

## for team 2, these need to be on the top half of the diagram
tm2 <- attack_rate$team == teams(x)[2]
attack_rate[tm2, c("x", "y")] <- dv_xy(attack_rate$start_zone, end = "upper")[tm2, ]

## plot this
ggplot(attack_rate, aes(x, y, fill = rate)) + geom_tile() + ggcourt(labels = teams(x)) +
  scale_fill_gradient2(name = "Attack rate")


## Example 2: map of starting and ending zones of attacks using arrows

## first tabulate attacks by starting and ending zone
attack_rate <- plays(x) %>% dplyr::filter(team == teams(x)[1] & skill == "Attack") %>%
  group_by(start_zone, end_zone) %>% tally() %>% ungroup

## convert counts to rates
attack_rate$rate <- attack_rate$n/sum(attack_rate$n)

## discard zones with zero attacks or missing location information
attack_rate <- attack_rate %>% dplyr::filter(rate>0 & !is.na(start_zone) & !is.na(end_zone))

## add starting x,y coordinates
attack_rate <- cbind(attack_rate,
     dv_xy(attack_rate$start_zone, end = "lower", xynames = c("sx","sy")))

## and ending x,y coordinates
attack_rate <- cbind(attack_rate,
     dv_xy(attack_rate$end_zone, end = "upper", xynames = c("ex","ey")))
```

```
## plot in reverse order so largest arrows are on the bottom
attack_rate <- attack_rate %>% dplyr::arrange(desc(rate))

p <- ggplot(attack_rate,aes(x,y,col = rate)) + ggcourt(labels = c(teams(x)[1],""))
for (n in 1:nrow(attack_rate))
    p <- p + geom_path(data = data.frame(x = c(attack_rate$sx[n], attack_rate$ex[n]),
                                         y = c(attack_rate$sy[n],attack_rate$ey[n]),
                                         rate = attack_rate$rate[n]),
       aes(size = rate), lineend = "round", arrow = arrow(ends = "last", type = "closed"))
p + scale_fill_gradient(name = "Attack rate") + guides(size = "none")

## End(Not run)
```

dv_xy2cone *Convert x, y coordinates to cones*

Description

Convert x, y coordinates to cones

Usage

```
dv_xy2cone(x, y = NULL, start_zones, force_center_zone = FALSE)
```

Arguments

x
: numeric: the x coordinate

y
: numeric: the y coordinate. If y is NULL, x will be treated as a grid index (see dv_index2xy)

start_zones
: numeric or character: the starting zone of each row (values 1-9, or "L", "M", "R")

force_center_zone
: logical: a vector indicating the rows that should be treated as center zone attacks regardless of their start_zone value (e.g. attacks by the setter). If FALSE, the start_zone value will be used. If provided as a single scalar value, this will be applied to all attacks

Value

A numeric vector giving the cone number

See Also

dv_xy2index, dv_index2xy, dv_cone2xy, dv_xy2zone, dv_xy2subzone

Examples

```
## Not run:

## a bunch of random points on and around the court
idx <- round(runif(100, min = 1, max = 10000))

## convert to cones, assuming a start_zone of "L"
cn <- dv_xy2cone(x = idx, start_zones = "M")

## generate the cone polygons for reference
cxy <- dv_cone_polygons("M")
cxyl <- dv_cone_polygons("M", end = "lower")

## plot
ggplot(cxy, aes(x, y, group = cone_number, fill = as.factor(cone_number))) +
  ## the cone polygons
  geom_polygon() + geom_polygon(data = cxyl) +
  ggcourt(labels = NULL) +
  ## and our points
  geom_point(data = dv_index2xy(idx) %>% mutate(cone_number = cn), shape = 21,
             colour = "black", size = 2)

## the points shoud be coloured the same as the cone polygons

## End(Not run)
```

---

dv_xy2subzone *Convert x, y coordinates to zones and subzones*

Description

Convert x, y coordinates to zones and subzones

Usage

```
dv_xy2subzone(x, y = NULL)
```

Arguments

x	numeric: the x coordinate
y	numeric: the y coordinate. If y is NULL, x will be treated as a grid index (see dv_index2xy)

Value

A tibble with columns zone and subzone

See Also

dv_xy2index, dv_index2xy, dv_cone2xy, dv_xy2zone

Examples

```
## Not run:

## a bunch of random points on and around the court
idx <- round(runif(100, min = 1, max = 10000))

## convert to zones
zn <- dv_xy2subzone(x = idx)

## or, equivalently, convert the index to xy values first
zn <- cbind(zn, dv_index2xy(idx))

## plot
ggplot(zn, aes(x, y, colour = as.factor(zone), shape = subzone)) + geom_point(size = 3) +
  ggcourt(labels = NULL)

## the points shoud be coloured by zone

## End(Not run)
```

dv_xy2zone *Convert x, y coordinates to zones*

Description

Convert x, y coordinates to zones

Usage

```
dv_xy2zone(x, y = NULL, as_for_serve = FALSE)
```

Arguments

x	numeric: the x coordinate
y	numeric: the y coordinate. If y is NULL, x will be treated as a grid index (see dv_index2xy)
as_for_serve	logical: if TRUE, treat the zones as if they refer to serving locations (i.e. zone 7 in between zones 5 and 6, and zone 9 in between zones 6 and 1)

Value

A numeric vector giving the zone number

See Also

dv_xy2index, dv_index2xy, dv_cone2xy, dv_xy2subzone

Examples

```
## Not run:

## a bunch of random points on and around the court
idx <- round(runif(100, min = 1, max = 10000))

## convert to zones
zn <- dv_xy2zone(x = idx)

## or, equivalently, convert the index to xy values first
idx_xy <- dv_index2xy(idx)
zn <- dv_xy2zone(x = idx_xy$x, idx_xy$y)

## plot
ggplot(idx_xy, aes(x, y, fill = as.factor(zn))) + geom_point(shape = 21) +
  ggcourt(labels = NULL)

## the points shoud be coloured by zone

## End(Not run)
```

findnext *Find each entry in y that follows each entry in x*

Description

Find each entry in y that follows each entry in x

Usage

```
findnext(x, y)
```

Arguments

x	numeric: vector
y	numeric: vector

Value

vector, each entry is the value in y that is next-largest to each corresponding entry in x

Examples

```
findnext(c(1,5,10),c(1,2,3,7,8,9))
```

`findprev` *Find each entry in y that precedes each entry in x*

Description

Find each entry in y that precedes each entry in x

Usage

```
findprev(x, y)
```

Arguments

x	numeric: vector
y	numeric: vector

Value

vector, each entry is the value in y that is next-smallest to each corresponding entry in x

Examples

```
findprev(c(1,5,10),c(1,2,3,7,8,9))
```

`find_first_attack` *Find first attacks by the receiving team (i.e. attacks associated with a serve reception)*

Description

Find first attacks by the receiving team (i.e. attacks associated with a serve reception)

Usage

```
find_first_attack(x)
```

Arguments

x	data.frame: the plays component of a datavolley object, as returned by `dv_read()`

Value

named list with components "ix" (logical indices into the x object where the row corresponds to a first attack in a rally), "n" (number of receptions for which there was a first attack by the receiving team), "n_win" (the number of winning first attacks), "win_rate" (number of winning first attacks as a proportion of the total number of first attacks).

See Also

dv_read plays

Examples

```
## Not run:
x <- dv_read(dv_example_file(), insert_technical_timeouts=FALSE)
## first attack win rate, by team
by(plays(x),plays(x)$team,function(z)find_first_attack(z)$win_rate)

## End(Not run)
```

find_match *Find a particular match in a list of datavolley objects*

Description

Find a particular match in a list of datavolley objects

Usage

```
find_match(match_id, x)
```

Arguments

match_id	string: match_id to find
x	list: list of datavolley objects as returned by dv_read

Value

numeric index of the match in the list

See Also

dv_read

find_player_name_remapping

Attempt to build a player name remapping table

Description

A player name can sometimes be spelled incorrectly, particularly if there are character encoding issues. This can be a particular problem when combining data from multiple files. This function will attempt to find names that have been misspelled and create a remapping table suitable to pass to `remap_player_names`. Player names will only be compared within the same team. Note that this function is unlikely to get perfect results: use its output with care.

Usage

```
find_player_name_remapping(x, distance_threshold = 3, verbose = TRUE)
```

Arguments

`x` datavolley: a datavolley object as returned by `dv_read`, or list of such objects

`distance_threshold` numeric: if two names differ by an amount less than this threshold, they will be treated as the same name

`verbose` logical: print progress to console as we go? Note that warnings will also be issued regardless of this setting

Value

data.frame with columns team, from, to

See Also

`remap_player_names`, `check_player_names`

Examples

```
## Not run:
  x <- dv_read(dv_example_file(), insert_technical_timeouts = FALSE)
  remap <- find_player_name_remapping(x)

## End(Not run)
```

`find_runs` *Generate information about runs of events*

Description

Find runs of events within a match. Typically, this function would be passed a subset of `plays(x)`, such as rows corresponding to serves. Runs that are terminated by the end of a set are not assigned a `run_length`.

Usage

```
find_runs(x, idvars = "team", within_set = TRUE)
```

Arguments

x	data.frame: a subset of the plays component of a datavolley object, as returned by dv_read()
idvars	character: string or character vector of variabe names to use to identify the entity doing the events
within_set	logical: only consider runs within a single set? If FALSE, runs that span sets will be treated as a single run

Value

A data.frame the same number of rows as `x`, and with columns `run_id` (the identifier of the run to which each row belongs), `run_length` (the length of the run), and `run_position` (the position of this row in its associated run).

See Also

`dv_read` `plays`

Examples

```
## Not run:
## find runs of serves
x <- dv_read(dv_example_file(), insert_technical_timeouts = FALSE)
serve_idx <- find_serves(plays(x))
serve_run_info <- find_runs(plays(x)[serve_idx,])
## distribution of serve run lengths
table(unique(serve_run_info[,c("run_id","run_length")])$run_length)

## End(Not run)
```

`find_serves` *Find serves*

Description

Find serves

Usage

```
find_serves(x)
```

Arguments

`x`	data.frame: the plays component of a datavolley object, as returned by `dv_read()`

Value

a logical vector, giving the indices of the rows of x that correspond to serves

See Also

`dv_read` `plays`

Examples

```
## Not run:
x <- dv_read(dv_example_file(), insert_technical_timeouts=FALSE)
serve_idx <- find_serves(plays(x))
## number of serves by team
table(plays(x)$team[serve_idx])

## End(Not run)
```

`fix_ace_evaluations` *Find aces that might not be marked as such*

Description

Some DataVolley files do not indicate serve aces with the skill evaluation "Ace". This function will search for winning serves, either with no reception or a reception error, and change their evaluation value to "Ace"

Usage

```
fix_ace_evaluations(x, rotation_error_is_ace = FALSE, verbose = TRUE)
```

Arguments

x datavolley: a datavolley object as returned by dv_read, or list of such objects

rotation_error_is_ace logical: should a rotation error on reception by the receiving team be counted as an ace?

verbose logical: print progress to console?

Value

datavolley object or list of such with updated evaluation values

See Also

dv_read

ggcourt *ggplot volleyball court*

Description

Volleyball court schematic suitable for adding to a ggplot

Usage

```
ggcourt(
  court = "full",
  show_zones = TRUE,
  labels = c("Serving team", "Receiving team"),
  as_for_serve = FALSE,
  show_zone_lines = TRUE,
  show_minor_zones = FALSE,
  show_3m_line = TRUE,
  grid_colour = "black",
  zone_colour = "grey70",
  minor_zone_colour = "grey80",
  fixed_aspect_ratio = TRUE,
  zone_font_size = 10,
  label_font_size = 12,
  label_colour = "black",
  court_colour = NULL,
  figure_colour = NULL,
  background_only = FALSE,
  foreground_only = FALSE,
  line_width = 0.5,
  xlim,
  ylim,
  ...
)
```

Arguments

`court` string: "full" (show full court) or "lower" or "upper" (show only the lower or upper half of the court)

`show_zones` logical: add numbers indicating the court zones (3m squares)?

`labels` string: labels for the lower and upper court halves (pass NULL for no labels)

`as_for_serve` logical: if TRUE and `show_zones` is TRUE, show zones as for serving. Only zones 1,5,6,7,9 are meaningful in this case

`show_zone_lines` logical: if FALSE, just show the 3m line. If TRUE, also show the 3m x 3m zones

`show_minor_zones` logical: add lines for the subzones (1.5m squares)?

`show_3m_line` logical: if TRUE, show the 3m (10ft) line

`grid_colour` string: colour to use for court sidelines, 3m line, and net

`zone_colour` string: colour to use for zone lines and labels

`minor_zone_colour` string: colour to use for minor zone grid lines

`fixed_aspect_ratio` logical: if TRUE, coerce the plotted court to be square (for a half-court plot) or a 2:1 rectangle (full court plot). Prior to package version 0.5.3 this was not TRUE by default

`zone_font_size` numeric: the font size of the zone labels

`label_font_size` numeric: the font size of the labels

`label_colour` string: colour to use for labels

`court_colour` string: colour to use for the court. If `NULL`, the court is only plotted with lines (no colour fill) and so the `figure_colour` will show through. Several special values are also supported here:

- `court_colour = "indoor"` can be used as a shortcut to set the court colour to orange, figure colour to blue, and lines and labels to white (similar to the typical indoor court colour scheme)
- `court_colour = "beach"` can be used as a shortcut to set the court and figure colour to a sandy-coloured yellow, lines and labels to black, and with the 3m line not shown by default
- `court_colour = "sand"` as for "beach" but with a sand texture image used as the court background

`figure_colour` string: colour to set the figure background to. If `NULL`, the background colour of the theme will be used (white, by default)

`background_only` logical: if `TRUE`, only plot the background elements (including general plot attributes such as the theme)

`foreground_only` logical: if `TRUE`, only plot the foreground elements (grid lines, labels, etc) `line_width` numeric: line width (passed as the size parameter to e.g. `ggplot2::geom_path`)

`xlim` numeric: (optional) limits for the x-axis

`ylim` numeric: (optional) limits for the y-axis

`...` : additional parameters passed to `ggplot2::theme_classic`

Details

The datavolley package uses the following dimensions and coordinates for plotting:

- the court is shown such that the sidelines are oriented vertically and the net is oriented horizontally
- the intersection of the left-hand sideline and the bottom baseline is at (0.5, 0.5)
- the intersection of the right-hand sideline and the top baseline is at (3.5, 6.5)
- the net intersects the sidelines at (0.5, 3.5) and (3.5, 3.5)
- the zones 1-9 (as defined in the DataVolley manual) on the lower half of the court are located at:
    1. (3, 1)
    2. (3, 3)
    3. (2, 3)
    4. (1, 3)
    5. (1, 1)
    6. (2, 1)
    7. (1, 2)
    8. (2, 2)
    9. (3, 2)
- the zones 1-9 (as defined in the DataVolley manual) on the upper half of the court are located at:
    1. (1, 6)
    2. (1, 4)
    3. (2, 4)
    4. (3, 4)
    5. (3, 6)
    6. (2, 6)
    7. (3, 5)
    8. (2, 5)
    9. (1, 5)

To get a visual depiction of this, try: `ggplot() + ggcourt() + theme_bw()`

Value

ggplot layer

See Also

dv_xy, dv_xy2index, dv_index2xy, dv_flip_xy

Examples

```
## Not run:
x <- dv_read(dv_example_file(), insert_technical_timeouts=FALSE)

library(ggplot2)
library(dplyr)

## Example 1: attack frequency by zone, per team

attack_rate <- plays(x) %>% dplyr::filter(skill == "Attack") %>%
  group_by(team, start_zone) %>% dplyr::summarize(n_attacks=n()) %>%
  mutate(rate=n_attacks/sum(n_attacks)) %>% ungroup

## add columns "x" and "y" for the x,y coordinates associated with the zones
attack_rate <- cbind(attack_rate, dv_xy(attack_rate$start_zone, end = "lower"))

## for team 2, these need to be on the top half of the diagram
tm2 <- attack_rate$team == teams(x)[2]
attack_rate[tm2, c("x", "y")] <- dv_xy(attack_rate$start_zone, end = "upper")[tm2, ]

## plot this
ggplot(attack_rate, aes(x, y, fill = rate)) + geom_tile() + ggcourt(labels = teams(x)) +
  scale_fill_gradient2(name = "Attack rate")


## Example 2: controlling layering
## use the background_only and foreground_only parameters to control the
##   order of layers in a plot

ggplot(attack_rate, aes(x, y, fill=rate)) +
  ## add the background court colours
  ggcourt(court_colour = "indoor", background_only = TRUE) +
  ## now the heatmap
  geom_tile() +
  ## and finally the grid lines and labels
  ggcourt(labels = teams(x), foreground_only = TRUE, court_colour = "indoor")


## Example 3: map of starting and ending zones of attacks using arrows

## first tabulate attacks by starting and ending zone
attack_rate <- plays(x) %>% dplyr::filter(team == teams(x)[1] & skill == "Attack") %>%
  group_by(start_zone, end_zone) %>% tally() %>% ungroup

## convert counts to rates
attack_rate$rate <- attack_rate$n/sum(attack_rate$n)

## discard zones with zero attacks or missing location information
```

```
attack_rate <- attack_rate %>% dplyr::filter(rate>0 & !is.na(start_zone) & !is.na(end_zone))

## add starting x,y coordinates
attack_rate <- cbind(attack_rate, dv_xy(attack_rate$start_zone, end = "lower",
                                        xynames = c("sx","sy")))

## and ending x,y coordinates
attack_rate <- cbind(attack_rate, dv_xy(attack_rate$end_zone, end = "upper",
                                        xynames = c("ex","ey")))

## plot in reverse order so largest arrows are on the bottom
attack_rate <- attack_rate %>% dplyr::arrange(desc(rate))

p <- ggplot(attack_rate, aes(x, y, col = rate)) + ggcourt(labels = c(teams(x)[1], ""))
for (n in 1:nrow(attack_rate))
    p <- p + geom_path(data = data.frame(x = c(attack_rate$sx[n], attack_rate$ex[n]),
                                         y = c(attack_rate$sy[n], attack_rate$ey[n]),
                                         rate = attack_rate$rate[n]),
                       aes(size = rate), lineend = "round",
                       arrow = arrow(length = unit(2, "mm"), type = "closed",
                                     angle = 20, ends = "last"))
p + scale_colour_gradient(name = "Attack rate") + guides(size = "none")

## End(Not run)
```

inspect *Convenience function for inspecting the plays component of a datavolley object*

Description

Convenience function for inspecting the plays component of a datavolley object

Usage

```
inspect(x, vars = "minimal", maxrows = 100, extra)
```

Arguments

x	datavolleyplays: the plays component of a datavolley object as returned by dv_read
vars	string: which variables to print? "minimal" set or "all"
maxrows	numeric: maximum number of rows to print
extra	character: names of any extra columns to include in the output

See Also

dv_read plays

Examples

```
## Not run:
  x <- dv_read(dv_example_file(), insert_technical_timeouts=FALSE)
  inspect(plays(x))

## End(Not run)
```

---

plays — *Extract the plays component from a datavolley object, or assign a new one*

Description

Extract the plays component from a datavolley object, or assign a new one

Usage

```
plays(x)

plays(x) <- value
```

Arguments

x	datavolley: a datavolley object as returned by dv_read
value	datavolleyplays: new data

Value

The plays component of x (a data.frame), or a modified version of x with the new plays component inserted

See Also

dv_read

Examples

```
## Not run:
  x <- dv_read(dv_example_file(), insert_technical_timeouts=FALSE)
  inspect(plays(x))

  p2 <- plays(x)
  plays(x) <- p2

## End(Not run)
```

play_phase — *Figure out the phase of play associated with each point*

Description

Phase is either "Serve", "Reception" (serve reception and the set and attack immediately following it, as well as the opposition block on that attack), or "Transition" (all play actions after that)

Usage

```
play_phase(x, method = "default")
```

Arguments

`x`	datavolleyplays: the plays component of a datavolley object as returned by `dv_read`
`method`	string: "default" (uses the `team_touch_id` and `skill` values to figure out phase), or "alt" (uses the sequences of `skill` values only. This is slower and probably less reliable, but will be more likely to give correct results in some situations (e.g. if the DataVolley file has been scouted in practice mode, and all actions have been assigned to the one team)

Value

character vector

See Also

`dv_read` `plays`

Examples

```
## Not run:
  x <- dv_read(dv_example_file(), insert_technical_timeouts = FALSE)
  px <- plays(x)
  px$phase <- play_phase(px)

## End(Not run)
```

print.summary.datavolley

Print method for summary.datavolley

Description

Print method for summary.datavolley

Usage

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

Arguments

x	summary.datavolley: a summary.datavolley object as returned by `summary.datavolley`
...	: additional arguments (currently these have no effect)

See Also

`summary.datavolley`

print.summary.datavolleylist

Print method for summary.datavolleylist

Description

Print method for summary.datavolleylist

Usage

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

Arguments

x	summary.datavolleylist: a summary.datavolleylist object, as returned by `dvlist_summary`
...	: additional arguments (currently these have no effect)

See Also

`dvlist_summary`

remap_player_info *Change player information*

Description

An experimental function to replace remap_player_names as a more comprehensive remapping of player attributes.

Usage

```
remap_player_info(x, remap)
```

Arguments

x	datavolley: a datavolley object as returned by dv_read, or list of such objects
remap	data.frame: data.frame of strings with columns team, name_from, and any of player_id, firstname, and lastname

Value

A datavolley object or list with corresponding player names changed

Examples

```
## Not run:
  x <- dv_read(dv_example_file(), insert_technical_timeouts = FALSE)
  x <- remap_player_info(x, data.frame(team = c("Nova KBM Branik", "Braslovče"),
                                   name_from = c("ELA PINTAR", "KATJA MIHALINEC"),
                            firstname = c("Ela", "Katja"), stringsAsFactors = FALSE))

## End(Not run)
```

remap_player_names *Change player names*

Description

A player name can sometimes be spelled incorrectly, particularly if there are character encoding issues. This can be a particular problem when combining data from multiple files. A player matching the team and from name entries in a row in remap is renamed to the corresponding to value. Alternatively, remap can be provided with the columns player_id and player_name: all player name entries associated with a given player_id will be changed to the associated player_name.

Usage

```
remap_player_names(x, remap)
```

Arguments

x	datavolley: a datavolley object as returned by dv_read, or list of such objects
remap	data.frame: data.frame of strings with columns team, from, and to

Value

A datavolley object or list with corresponding player names changed

See Also

dv_read, check_player_names, find_player_name_remapping

Examples

```
## Not run:
  x <- dv_read(dv_example_file(), insert_technical_timeouts = FALSE)
  x <- remap_player_names(x, data.frame(team = c("Nova KBM Branik", "Braslovče"),
                                        from = c("ELA PINTAR", "KATJA MIHALINEC"),
                                        to = c("Ela PINTAR", "Katja MIHALINEC"),
                                        stringsAsFactors = FALSE))

  x <- remap_player_names(x, data.frame(player_id = c("id1", "id2"),
                                      player_name = c("name to use 1", "name to use 2"),
                                        stringsAsFactors = FALSE))

## End(Not run)
```

remap_team_names *Change team names*

Description

A team name can sometimes be spelled incorrectly, particularly if there are character encoding issues. This can be a particular problem when combining data from multiple files. If a team name matches the from entry and/or its ID matches the team_id entry in a row in remap, the team will be renamed to the corresponding to value and/or its ID changed to the corresponding to_team_id value.

Usage

```
remap_team_names(x, remap, fixed = TRUE)
```

Arguments

x	datavolley: a datavolley object as returned by dv_read, or list of such objects
remap	data.frame: data.frame of strings with one or both columns from and team_id, and one or both columns to and to_team_id
fixed	logical: treat the from and team_id entries as fixed strings? If fixed is FALSE they will be treated as regular expressions

Value

datavolley object or list with corresponding team names changed

See Also

dv_read

Examples

```
## Not run:
  x <- dv_read(dv_example_file(), insert_technical_timeouts = FALSE)
  summary(x)

  ## rename a team based just on team name
  summary(remap_team_names(x, data.frame(from="Nova KBM Branik", to="NKBM Branik")))

  ## rename a team based on team name and ID
 summary(remap_team_names(x, data.frame(from="Nova KBM Branik", to="NKBM Branik", team_id="MB4")))

## End(Not run)
```

serve_win_points *Find serve win points*

Description

Find points in which the serving team wins the point. Serve win rate is the fraction of serves won by the serving team.

Usage

```
serve_win_points(x, return_id = FALSE)
```

Arguments

`x` data.frame: the plays component of a datavolley object, as returned by `dv_read()`

`return_id` logical: include the match_id and point_id of all serve win points in the returned object?

Value

named list with components "ix" (logical indices of serves corresponding to serve win points in the x object), "n" (number of serve win points in x), "rate" (serve win rate from x). If `return_id` is TRUE, also return a component "id" (a data.frame containing the match_id and point_id of all serve win points)

See Also

dv_read plays

Examples

```
## Not run:
x <- dv_read(dv_example_file(), insert_technical_timeouts=FALSE)
serve_idx <- find_serves(plays(x))
swp <- serve_win_points(plays(x))
## number of serves by team
table(plays(x)$team[serve_idx])
## number of points won on serve by team
table(plays(x)$team[serve_idx & swp$ix])

## End(Not run)
```

skill_evaluation_decoder

Translate skill evaluation codes into meaningful summary phrases

Description

If your DataVolley files use evaluation codes differently to those coded here, you will need to supply a custom skill_evaluation_decode function to dv_read

Usage

```
skill_evaluation_decoder(style = "default")
```

Arguments

style	string: currently "default" (following the standard definitions described in the DataVolley manual) or "volleymetrics" (per the conventions that VolleyMetrics use)

Value

function. This function takes arguments skill, evaluation_code, and show_map and returns a string giving the interpretation of that skill evaluation code

See Also

dv_read

Examples

```
sd <- skill_evaluation_decoder()
sd("S","#")
sd(show_map=TRUE)
```

summary.datavolley *A simple summary of a volleyball match*

Description

A simple summary of a volleyball match

Usage

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

Arguments

`object`	datavolley: datavolley object as returned by `dv_read`
`...`	: additional arguments (currently these have no effect)

Value

list of summary items

See Also

`dv_read`

Examples

```
x <- dv_read(dv_example_file(), insert_technical_timeouts=FALSE)
summary(x)
```

teams *Get team names and IDs from datavolley object*

Description

Get team names and IDs from datavolley object

Usage

```
teams(x)

home_team(x)

home_team_id(x)

visiting_team(x)

visiting_team_id(x)
```

Arguments

x	datavolley or data.frame: a datavolley object as returned by `dv_read`, or the plays component of that object

Value

character vector of team names or IDs

See Also

`dv_read`

Examples

```
## Not run:
  x <- dv_read(dv_example_file(), insert_technical_timeouts = FALSE)
  teams(x)
  home_team_id(x)

## End(Not run)
```

Index