Package 'StereoMorph'

April 7, 2015

Date 2015-04-06

Title Stereo Camera Calibration and Reconstruction

Description Functions for the collection of 3D points and curves using a stereo camera setup.

Version 1.4

Depends R (>= 2.11.0)

Imports bezier (>= 1.1), grid, rjson, shiny, Rcpp (>= 0.9.9), jpeg, tiff, png

LinkingTo Rcpp

Suggests rgl

Author Aaron Olsen, Annat Haber

Maintainer Aaron Olsen <aolsen@uchicago.edu>

Repository CRAN

URL http://home.uchicago.edu/~aolsen/software/stereomorph.shtml

License GPL (>= 2)

NeedsCompilation yes

Date/Publication 2015-04-07 09:10:09

R topics documented:

`StereoMorph-package` *Stereo Camera Morphometrics*

Description

StereoMorph provides functions for the collection of 3D points and curves using a stereo camera setup. **StereoMorph** can also be used for collecting 2D shape data from photographs. Please see StereoMorph 10-step Tutorial for a step-by-step tutorial on how to use **StereoMorph**.

Details

Package:	StereoMorph
Type:	Package
Version:	1.4
Date:	2015-04-06
License:	GPL-2

Author(s)

Aaron Olsen, Annat Haber

Maintainer: Aaron Olsen <aolsen@uchicago.edu>

See Also

svgViewR, bezier

---

alignLandmarksToMidline

Aligns bilateral landmarks to the midline plane

Description

This function aligns a set of bilateral landmarks to the midline plane. Midline landmarks and the mean position of bilateral landmarks are both used to define the midline plane.

Usage

```
alignLandmarksToMidline(lm.matrix, left = '(_L|_l|_left|_LEFT)([0-9]*$)',
                        right = '(_R|_r|_right|_RIGHT)([0-9]*$)',
                        left.remove = '\\2', right.remove = '\\2')

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

Arguments

lm.matrix	a 2D or 3D matrix with landmark names as row names.
left	a regular expression to identify left landmarks in the row names of lm.matrix.
right	a regular expression to identify right landmarks in the row names of lm.matrix.

left.remove	an expression for input to the gsub() function indicating which element of left in parentheses should be removed to create a landmark name that is not side-specific (see "Details").
right.remove	an expression for input to the gsub() function indicating which element of right in parentheses should be removed to create a landmark name that is not side-specific (see "Details").
object	a list of class "alignLandmarksToMidline" (output of this function).
...	further arguments passed to or from other methods.

Details

Currently, the function only accepts left/right designations by matching a regular expression to the landmark name. This is preferable in that it allows for easier match up between bilateral landmarks (based on their common name without a side annotation). The default regular expression identifies left landmarks by a name ending in "_L", "_l", "_left" or "_LEFT", optionally followed by numbers. For example, "hamulus_left", "hamulus_L" and "zymgomatic_arch_l012" would all be identified as landmarks on the left side. Similarly, "hamulus_right", "hamulus_R" and "zymgomatic_arch_r012" would all be identified as landmarks on the right side. Landmarks not identified as left or right are assumed to fall on the midline.

In order to find corresponding left and right landmarks, the function requires the left.remove and right.remove arguments. The left.remove and right.remove arguments are passed to the base function gsub() as the replacement argument. This is used to generate a landmark name that is not side-specific. For example, "hamulus_left" and "zymgomatic_arch_l012" would become "hamulus" and "zymgomatic_arch012". These will be reverted to their original names at return.

Once corresponding right and left landmarks have been identified, the function finds the mean positions of all bilateral landmarks and the positions of all midline landmarks. These points are used to define the midline. After alignment, the specimen will have the midline axis as the last column (z in 3D, y in 2D), the longest non-midline axis as the first column (x in 3D), and the second non-midline axis as y for 3D. No further rotation and reflection is done, therefore the specimen may be facing any direction along each of the axes.

This function returns the aligned landmarks and an error vector, midline.error. This is a vector of the squared z-coordinate of the midline landmarks (the distance between each midline landmark and the midline plane). If reflectMissingLandmarks was called on the landmarks prior to alignLandmarksToMidline() with average equal to TRUE, then all of the midline points will fall exactly along the midline. Thus, the error vector will consist entirely of zeros (or near-zero values).

For a step-by-step tutorial on how to use alignLandmarksToMidline() see Unifying, reflecting and aligning landmarks.

Value

a list of class "alignLandmarksToMidline" with the following elements:

lm.matrix	a 2D or 3D matrix of landmarks aligned to the midline.
midline.error	a vector of the errors (distances) between each midline landmark and the midline plane.

Note

This function was modified by A Olsen from the R function `AMP()` written by A Haber.

Author(s)

Annat Haber, Aaron Olsen

See Also

`readLandmarksToMatrix`

Examples

```
## FIND THE FILE DIRECTORY FOR EXTRA R PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## GET LANDMARKS
file <- paste0(fdir, "lm_3d_unify.txt")

## LOAD FILES INTO A MATRIX
lm.matrix <- readLandmarksToMatrix(file=file, row.names=1)

## ALIGN TO MIDLINE
align_landmarks <- alignLandmarksToMidline(lm.matrix=lm.matrix)

## PRINT SUMMARY OF ERRORS
print(summary(align_landmarks))
```

---

avectors *Computes the angle between two vectors*

Description

This function returns the angle (in radians) between two vectors. The vectors can be of any dimension.

Usage

```
avectors(u, v)
```

Arguments

u	a vector
v	a vector

Value

the angle (in radians) between the two input vectors.

Author(s)

Aaron Olsen

Examples

```
## THE ANGLE BETWEEN TWO 2D, ORTHOGONAL VECTORS
## VALUE IS EQUAL TO asin(1/sqrt(2))
u <- c(0, 1)
v <- c(1, 0)
avectors(u, v)

## THE ANGLE BETWEEN TWO 3D VECTORS
## VALUE IS EQUAL TO asin(sqrt(2)/sqrt(3))
u <- c(1, 1, 1)
v <- c(0, 1, 0)
avectors(u, v)
```

---

cprod *Computes the cross product of two vectors*

Description

Returns the cross product of two vectors using either the right-hand or left-hand convention.

Usage

```
cprod(u, v, h = "right")
```

Arguments

u	a vector of length 3
v	a vector of length 3
h	whether the right-hand, "right", or left-hand, "left", convention is to be used. The right-hand convention is default.

Details

The cross product vector is a vector orthogonal to the two input vectors.

Value

the cross product vector (also of length 3)

Author(s)

Aaron Olsen

Examples

```
## DEFINE TWO 3D VECTORS
u <- c(1, 0, 0)
v <- c(0, 1, 0)

## FIND THE CROSS PRODUCT
cprod(u, v)
```

`digitizeImage` *Opens the StereoMorph Digitizing App*

Description

This function opens an application in the user's default web browser for manually digitizing landmarks and Bezier curves from photographs.

Usage

```
digitizeImage(image.file, landmarks.file=NULL, control.points.file=NULL,
              curve.points.file=NULL, shapes.file=NULL,
              landmarks.ref=NULL, curves.ref=NULL, image.id=NULL,
              landmark.color.blur = 'blue', landmark.color.focus = 'green',
              curve.color.blur = 'purple', control.point.color.blur = 'purple',
              control.point.color.focus = 'red', landmark.radius = 4,
              control.point.radius = 4, marker.stroke.width = 2, app.dir=NULL)
```

Arguments

`image.file` a file path to the image or images to be digitized. This can be a folder containing one or more images or a vector of file paths of one or more images.

`landmarks.file` a file path indicating where the landmarks should be loaded from and/or saved to. If `landmark.file` is a folder, then landmark files are assumed to have the same name as the images (with a .txt extension). Alternatively, this can be a vector of file paths to .txt files. If `landmarks.file` is a vector of file paths, the number of paths must equal the number of images to be digitized. If not collecting landmarks, this argument can be omitted.

`control.points.file` a file path indicating where the Bezier control points should be loaded from and/or saved to. If `control.points.file` is a folder, then control point files are assumed to have the same name as the images (with a .txt extension). Alternatively, this can be a vector of file paths to .txt files. If `control.points.file` is a vector of file paths, the number of paths must equal the number of images to be digitized. If not collecting curves, this argument can be omitted.

`curve.points.file`
: a file path indicating where the Bezier curve points should be saved to. If `curve.points.file` is a folder, then curve point files are assumed to have the same name as the images (with a .txt extension). Alternatively, this can be a vector of file paths to .txt files. If `curve.points.file` is a vector of file paths, the number of paths must equal the number of images to be digitized. If not collecting curve points, this argument can be omitted.

`shapes.file`
: a file path or folder indicating where shape files should be saved. This is a new format that is currently only intended to be used when collecting 2D data. The other input types (`landmarks.file`, `control.points.file`, `curve.points.file`) will be phased out in future updates and replaced with this format.

`landmarks.ref`
: landmarks to be digitized. This can either be a file path to a .txt file containing the landmarks (listed in a single column, each on a separate line) or a vector of landmark names.

`curves.ref`
: curves to be digitized. For each curve, the name of the curve, the starting point and the ending point must be specified. `curves.ref` can either be a three-column matrix (with the curve name in the first column, the starting point in the second and the end point in the third) or a file path to a .txt file containing a three-column curve reference matrix. If a file path, the file should have no header and tab-separated row values.

`image.id`
: Image IDs to be saved with each image. These will be used to reference shape data in the output of `readShapes`. If `NULL`, the filenames of the images will be used (without the file extension).

`landmark.color.blur`
: The color of an unselected landmark. It might be necessary to change if the background color is close to the default. Colors must be valid SVG color names or codes (e.g. "hotpink", "#4B0082", etc.). A web-search for "SVG color codes" will indicate several possible options.

`landmark.color.focus`
: The color of a selected landmark. See `landmark.color.blur`.

`curve.color.blur`
: The color of digitized curves. A different color for a selected curve is not yet supported. See `landmark.color.blur`.

`control.point.color.blur`
: The color of an unselected control point. See `landmark.color.blur`.

`control.point.color.focus`
: The color of a selected control point. See `landmark.color.blur`.

`landmark.radius`
: The radius of digitized landmarks.

`control.point.radius`
: The radius of the Bezier control points.

`marker.stroke.width`
: The thickness of the lines used to draw the landmarks and control points.

`app.dir`
: Changes the shiny app directory for debugging.

Details

This function opens a digitizing app in the user's default browswer and allows for the digitization of landmarks and Bezier curves from photographs. Although the app runs in a web browser, the user does not have to be connected to the internet as the app runs on a local server. The R package 'shiny' handles the communication between the browser and the R console. Safari, Chrome and Opera all provide full compatibility with the apps's features.

One or more images can be uploaded to the app. If more than one image is uploaded (via `image.file`), users can switch between images within the app by clicking the "Previous Image" and "Next Image" buttons in the app control panel. The app can be used to digitize landmarks only, curves only or both landmarks and curves. To only digitize landmarks omit the arguments `control.points.file`, `curve.points.file` and `curves.ref`. To only digitize curves, omit the arguments `landmarks.file` and `landmarks.ref`. If the files input via `landmarks.file` and `control.points.file` already contain landmarks or control points, these will be loaded in the app and can be modified and re-saved.

The app can save both Bezier control points and Bezier curve points. The former are the points that can be added and moved by the user while digitizing; these are used to change the shape of the curve but are not the curve points themselves. The latter are hundreds of points at single-pixel spacing that actually describe the curve and are appropriate for subsequent shape analysis. These are generated by the `pointsOnBezier()` function in the R package 'bezier'. While it is necessary to specify `control.points.file` when digitizing curves, `curve.points.file` is optional.

For a step-by-step tutorial on how to open and use the StereoMorph Digitizing App and download the StereoMorph Tutorial folder, see Digitizing with StereoMorph. Additional instructions can be found by clicking on 'Help' in the right upper corner of the app (or viewing the help file here).

Value

NULL

Author(s)

Aaron Olsen

See Also

`readShapes`

Examples

```
## Not run:
## THE FOLLOWING EXAMPLES CAN BE RUN
## BUT USERS MUST FIRST DOWNLOAD THE STEREOMORPH TUTORIAL FOLDER
## SEE DETAILS ABOVE

## TO DIGITIZE ONLY LANDMARKS IN A SINGLE IMAGE
digitizeImage(
   image.file = "Object images/obj1_a1_v1.JPG",
   landmarks.file = "Landmarks 2D/obj1_a1_v1.txt",
   landmarks.ref = "landmarks_ref.txt")
```

```
## TO DIGITIZE ONLY LANDMARKS IN MULTIPLE IMAGES
digitizeImage(
   image.file = c("Object images/obj1_a1_v1.JPG", "Object images/obj1_a1_v1.JPG"),
   landmarks.file = c("Landmarks 2D/obj1_a1_v1.txt", "Landmarks 2D/obj1_a1_v1.txt"),
   landmarks.ref = "landmarks_ref.txt")

## OR ALTERNATIVELY, IF "OBJECT IMAGES" IS A FOLDER CONTAINING IMAGES
digitizeImage(
   image.file = "Object images",
   landmarks.file = "Landmarks 2D",
   landmarks.ref = "landmarks_ref.txt")

## DIGITIZING CURVES
digitizeImage(
   image.file = "Object images",
   control.points.file = "Control points",
   curve.points.file = "Curve points 2D",
   curves.ref = "curves_ref.txt")

## DIGITIZING LANDMARKS AND CURVES
digitizeImage(
   image.file = "Object images",
   landmarks.file = "Landmarks 2D",
   control.points.file = "Control points",
   curve.points.file = "Curve points 2D",
   landmarks.ref = "landmarks_ref.txt",
   curves.ref = "curves_ref.txt")

## End(Not run)
```

`distanceGridUnits` *Returns the distances between pairs of points on a square grid*

Description

This function returns the distances in grid units between pairs of points on a square grid. This function is used in testing the accuracy of a calibration by comparing theoretical distances among grid points to measured distances.

Usage

```
distanceGridUnits(pairs, nx)
```

Arguments

`pairs`	a two-column matrix specifying the pairs of points between which the distance is to be found.
`nx`	the number of points in the first dimension along which grid points are counted (see "Details").

Details

This function returns the distances in grid units between pairs of points on a square grid. Thus, adjoining points in the same row or column would be separated by a distance of one. The returned distances can then be multiplied by the grid square size to obtain the distances between pairs of points in real-world units (e.g. mm).

The input `pairs` is a two-column matrix specifying the pairs of points, with the first column corresponding to one point and the second column to the other. The numbers in `pairs` are indices of grid points (not point coordinates themselves). The assumed numbering scheme for the grid points is as follows: the points are first numbered across the first dimension (of length `nx`) and then along a second dimension. For example, on a 5x4 grid points 1-5 would be across the first row, 6-10 across the second row, etc. For each row the point numbering starts in the same column. For `distanceGridUnits()` the number of columns does not need to be specified since this can be found from the point index in `pairs`. See "Examples" for an explanation of the numbering scheme.

Value

a vector of the distances between the specified pairs of grid points.

Author(s)

Aaron Olsen

See Also

`dltTestCalibration`

Examples

```
## INDICES OF POINT PAIRS ON A GRID WITH 5 ROWS
pairs <- matrix(c(1,1, 5,10, 6,16, 1,20), nrow=4, ncol=2, byrow=TRUE)

## FIND THE DISTANCE BETWEEN PAIRS OF POINTS IN GRID UNITS
## NOTE LAST DISTANCE IS 5 BECAUSE IT IS A 3,4,5-TRIANGLE
distanceGridUnits(pairs, nx=5)

## FOR ILLUSTRATION, HERE IS A GRID WITH 5 ROWS AND 4 COLUMNS
xy <- cbind(rep(0:4, 4), c(rep(0, 5), rep(1, 5), rep(2, 5), rep(3, 5)))

## PLOT THESE POINTS
plot(xy)

## PLOT LINE SEGMENTS CONNECTING THE PAIRS ABOVE
segments(x0=xy[pairs[, 1], 1], y0=xy[pairs[, 1], 2],
  x1=xy[pairs[, 2], 1], y1=xy[pairs[, 2], 2],
  col=c('blue', 'red', 'purple', 'green'))
```

`distancePointToLine` *Finds the minimum distance(s) between point(s) and a line*

Description

Finds the minimum distance between a point and a line or multiple points and a line in two or three dimensions.

Usage

```
distancePointToLine(p, l1, l2 = NULL)
```

Arguments

`p`	a vector of a single point or a matrix of multiple points
`l1`	a vector describing a point on a line or a list with line constants
`l2`	if `l1` is a point, a second point on a line

Details

If `p` is a vector, the function returns the distance between a point and the line input. If `p` is a matrix, the function returns the distance between each point in the matrix (defined by each row) and the line input. If `p` is a vector, the length must be 2 or 3 (2D or 3D, respectively). If `p` is a matrix, the number of columns must be 2 or 3 (2D or 3D, respectively).

The line input can be defined using one of three standard ways: two points on the line, 'm' and 'b' constants and direction numbers (a vector parallel to the line). If `l1` is a vector, this is taken as one point on the line and `l2` must be a second point on the line. If `l1` is a list, the named objects must correspond to one of these three line definitions. Two points on the line are defined as `l1$l1` and `l1$l2`. 'm' and 'b' are defined as `l1$m` and `l1$b`. And the direction numbers 'abc' are defined as `l1$a`, `l1$b` and `l1$c`.

Value

a vector of distance(s)

Author(s)

Aaron Olsen

See Also

`orthogonalProjectionToLine`

Examples

```
## FIND THE DISTANCE BETWEEN A 2D POINT AND A LINE DEFINED BY A SLOPE AND Y-INTERCEPT
distancePointToLine(p=c(0, 2), l1=list(m=0, b=1))

## FIND THE DISTANCE BETWEEN A 2D POINT AND A LINE DEFINED BY TWO POINTS ON THE LINE
distancePointToLine(p=c(0, 5), l1=list(l1=c(2, 4), l2=c(2, 1)))

## FIND THE DISTANCE BETWEEN MULTIPLE 2D POINTS AND A LINE DEFINED BY A SLOPE AND Y-INTERCEPT
p <- matrix(c(0, 0, 1, 1, 2, 2), nrow=3, ncol=2, byrow=TRUE)
distancePointToLine(p=p, l1=list(m=0, b=1))

## FIND THE DISTANCE BETWEEN MULTIPLE 2D POINTS AND A LINE DEFINED BY DIRECTION NUMBERS
p <- matrix(c(0, -1.5, 1, -2, 2, 2), nrow=3, ncol=2, byrow=TRUE)
distancePointToLine(p=p, l1=list(a=1, b=2, c=3))

## FIND THE DISTANCE BETWEEN A 3D POINT AND A LINE DEFINED BY TWO POINTS ON THE LINE
## HERE THE DISTANCE IS EQUAL TO sqrt(2)
distancePointToLine(p=c(1, 1, 1), l1=c(0, 0, 0), l2=c(1, 0, 0))

## FIND THE DISTANCE BETWEEN MULTIPLE 3D POINTS AND A LINE DEFINED BY TWO POINTS ON THE LINE
p <- matrix(c(0, 0, 0, 1, 1, 1, 2, 2, 2), nrow=3, ncol=3, byrow=TRUE)
distancePointToLine(p=p, l1=list(l1=c(0, 0, 0), l2=c(1, 0, 0)))
```

---

distancePointToPoint *Finds the distance between two points or sets of points*

Description

Finds the distance betweeen two single points, the distances between one point and a set of points, or the distances between two point sets. Points can be of any number of dimensions.

Usage

```
distancePointToPoint(p1, p2 = NULL)
```

Arguments

p1	a vector of a single point or a matrix of one or multiple points
p2	a vector of a single point or a matrix of one or multiple points. If NULL, the function either returns the distance of p1 from the origin or the distances between subsequent values of p1.

Details

If p1 is a single point and p2 is a single point then the function returns the distance between these two points. If either p1 or p2 is a single point and the other is a matrix of multiple points then the function returns a vector of the distances between the single point and each of the multiple points. If both p1 and p2 are matrices of multiple points, then the function returns a vector of the distances

between the points in each corresponding row. If p1 and p2 are both matrices, the matrix dimensions must match.

If p2 is NULL, then distancePointToPoint() returns the distance between consecutive points in p1. If p1 is a vector, the function returns the absolute difference between consecutive values of p1 (interpoint distances along a single dimension). If p1 is a matrix, then the function returns the distance between the point in each row of p1 and its subsequent row. This can be used to return the interpoint distances along a curve defined as a matrix of points.

Value

a vector of distance(s)

Author(s)

Aaron Olsen

See Also

distancePointToLine

Examples

```
## FIND THE DISTANCE BETWEEN TWO, 2D POINTS
## VALUE IS sqrt(2)
distancePointToPoint(p1=c(0, 0), p2=c(1, 1))

## FIND THE DISTANCE BETWEEN A 2D POINT AND MULTIPLE 2D POINTS
p1 <- c(0, 0)
p2 <- matrix(c(1, 1, 2, 2, 3, 3), nrow=3, ncol=2, byrow=TRUE)
distancePointToPoint(p1=p1, p2=p2)

## FIND THE DISTANCE BETWEEN TWO SETS OF 2D POINTS
p1 <- matrix(c(0, 0, 1, 1, 2, 2), nrow=3, ncol=2, byrow=TRUE)
p2 <- matrix(c(1, 1, 2, 2, 3, 3), nrow=3, ncol=2, byrow=TRUE)
distancePointToPoint(p1=p1, p2=p2)

## FIND THE DISTANCE BETWEEN A 3D POINT AND MULTIPLE 3D POINTS
p1 <- c(0, 0, 0)
p2 <- matrix(c(1, 1, 1, 2, 2, 2, 3, 3, 3), nrow=3, ncol=3, byrow=TRUE)
distancePointToPoint(p1=p1, p2=p2)

## FIND THE DISTANCE BETWEEN CONSECUTIVE VALUES IN A VECTOR
distancePointToPoint(p1=c(1, 2, 4, 7))

## FIND THE DISTANCE BETWEEN CONSECUTIVE 2D POINTS IN A MATRIX
## HERE, WE FIND THE DISTANCE BETWEEN THE POINT c(0, 0) AND c(1, 1), WHICH IS sqrt(2)
distancePointToPoint(p1=matrix(c(0, 0, 1, 1), nrow=2, ncol=2, byrow=TRUE))

## FIND THE DISTANCE BETWEEN CONSECUTIVE 2D POINTS IN A MATRIX, WITH MORE POINTS
## HERE, WE ADD TWO MORE POINTS TO THE PREVIOUS EXAMPLE: c(2, 2) AND c(3, 3)
## THE DISTANCE BETWEEN EACH CONSECUTIVE PAIR OF POINTS IS sqrt(2)
```

```
distancePointToPoint(p1=matrix(c(0, 0, 1, 1, 2, 2, 3, 3), nrow=4, ncol=2, byrow=TRUE))

## FIND THE DISTANCE BETWEEN CONSECUTIVE 3D POINTS IN A MATRIX
distancePointToPoint(p1=matrix(c(0, 0, 0, 1, 1, 1), nrow=2, ncol=3, byrow=TRUE))

## FIND THE DISTANCE BETWEEN CONSECUTIVE 4D POINTS IN A MATRIX
distancePointToPoint(p1=matrix(c(0, 0, 0, 0, 1, 1, 1, 1), nrow=2, ncol=4, byrow=TRUE))
```

`dltCalibrateCameras` *Finds the optimized DLT coefficients for a stereo camera setup*

Description

This function uses the corners from a grid positioned in several different orientations within a stereo camera setup to estimate the DLT calibration coefficients that minimize reconstruction error.

Usage

```
dltCalibrateCameras(coor.2d, nx, grid.size, c.run = FALSE, reduce.grid.dim = 3,
                fit.min.break = 1, nlm.iter.max.init = 100, objective.min.init = 10,
                    nlm.eval.max = 350, nlm.iter.max = 250, nlm.calls.max = 100,
                     objective.min = 1, grid.incl.min=2, start.param=NULL,
                     print.progress = FALSE)

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

Arguments

`coor.2d`	a four-dimensional array of grid points. The first two dimensions correspond to each matrix of grid points, the third corresponds to each grid position/orientation and the fourth corresponds to each camera view. Can be read from file by the function `readCheckerboardsToArray`.
`nx`	the number of points along the first dimension (e.g. this would be the number of points in each row if points in `coor.2d` are listed first by row). The number of points along the second dimension is calculated based on the total number of points per view and orientation.
`grid.size`	the size of the grid squares in real-world units (e.g. millimeters).
`c.run`	a logical indicating whether a second optimization should be performed on the calibration coefficients.
`reduce.grid.dim`	a numeric indicating the number of grid points along each dimension for each grid after resampling. The total number of resampled points is `reduce.grid.dim^2`. Resampling can be turned off by setting this to `0` or `FALSE`. The default is recommended. `reduce.grid.dim` must be greater than two.

`fit.min.break` passed to `resampleGridImagePoints()`. A minimum returned by `nlminb()` (indicating goodness of fit in pixel coordinates) at which `resampleGridImagePoints()` will stop iterating to find a better fit for each checkerboard grid. Ignored if `reduce.grid.dim` is `0` or `FALSE`.

`nlm.iter.max.init` The maximum number of iterations to be performed by `nlminb()` during initial coefficient optimization, passed as a control parameter to `nlminb()`. These are the number of iterations for an initial determination of whether the function is likely to converge on the correct estimate.

`objective.min.init` The objective used during the initial coefficient optimization, passed as a control parameter to `nlminb()`, to determine whether the function is close to convergence.

`nlm.eval.max` The maximum number of evaluations to be performed by `nlminb()` during primary coefficient optimization, passed as a control parameter to `nlminb()`. Keeping this value as low as possible without excluding actual convergence speeds performance of the function by preventing the function from stalling far from the optimal values.

`nlm.iter.max` The maximum number of iterations to be performed by `nlminb()` during primary coefficient optimization, passed as a control parameter to `nlminb()`. Keeping this value as low as possible without excluding actual convergence speeds performance of the function by preventing the function from stalling far from the optimal values.

`nlm.calls.max` The maximum number of different sets of random starting parameters to use during coefficient optimization. This parameter cannot exceed 576.

`objective.min` The expected mean reconstruction error when optimizing the calibration coefficients (the minimum, or `objective` value returned by `nlminb()`). A value between 0.7 and 3 should be reasonable.

`grid.incl.min` The minimum number of grids to include during coefficient optimization.

`start.param` An set of fixed starting parameters to be used during coefficient optimization. This parameter is intended primarily for debugging.

`print.progress` a logical indicating whether the progress of the function should be printed while running. This includes the error in grid re-sampling, an iteration count during optimization and other outputs relating to the optimization.

`object` a list of class `"dltCalibrateCameras"` (the output of `dltCalibrateCameras()`).

`...` further arguments passed to or from other methods.

Details

Calibration is the most challenging step in stereo camera data collection. Most fundamentally, DLT calibration requires a set of 3D coordinates and their corresponding 2D pixel coordinates in each camera view in order to derive calibration coefficients (see `dltCoefficients`). These coefficients can then be used to reconstruct any point in 3D given its 2D pixel coordinates in two or more camera views. DLT calibration has traditionally been done using a "calibration object", typically a 3D box-shaped structure filled with markers at known 3D positions. Such objects require the use of

high precision machining in order to achieve an accurate calibration and the calibration points are usually digitized manually.

The `dltCalibrateCameras()` function provides a camera calibration routine that is easier to implement and potentially more accurate. This function uses the corners from a grid positioned in several different orientations within the calibration space to estimate the DLT calibration coefficients that minimize reconstruction error. The easiest method for obtaining these corner points is to print a checkerboard pattern (using `drawCheckerboard`), attach the pattern to flat, hard surface and use `findCheckerboardCorners` to automatically extract the pixel coordinates of the internal corners. For a step-by-step tutorial on how to create a checkerboard pattern, see Creating a checkerboard pattern.

The grid pattern should be photographed in at least four different positions and orientations spanning the volume to be calibrated (the tutorial files loaded with StereoMorph include eight different positions). Using only a couple of positions will result in uneven sampling of the calibration volume causing larger errors in some regions relative to others. Additionally, using only a single orientation of the checkerboard will produce higher errors along a particular dimension relative to the others. Once the pixel coordinates of the grid points (e.g. the internal corners of the checkerboard pattern) have been extracted from all of the calibration images, they should be read into an array using `readCheckerboardsToArray`. This function allows for the point order to be reversed along rows, columns or both. If one of the cameras views the pattern upside down relative to another camera or if the pattern is in a different orientation, the grid points may be extracted in a different order. This can be fixed using the `row.reverse` and `col.reverse` arguments in `readCheckerboardsToArray`. It is **essential** that the grid points extracted from each camera view correspond to each other row-by-row or else the calibration will not work.

`dltCalibrateCameras()` first calls `resampleGridImagePoints` to downsample the number of grid points. `reduce.grid.dim` is the downsample number (the default is 3, meaning 3x3 or nine points per grid). Downsampling can be turned off by setting `reduce.grid.dim` to `0`, although this is not recommended as it will increase run-time substantially without increasing accuracy. A camera perspective model is fit to the full point set such that the number of points input to the coefficient optimization can be reduced (thereby reducing run-time) without losing any relevant information (see `resampleGridImagePoints`). If `print.progress` is set to `TRUE`, the mean and maximum fit error is printed for each input grid. As the fitting does not take into account lens distortion, high fit errors may indicate large distortional effects.

Since each checkerboard grid has been photographed in an arbitrary position and orientation, the 3D coordinates of the grid points are unknown. However, if the first grid is fixed, each additional grid can be described by applying six transformation parameters relative to the first (three translational and three rotational). Using the reduced grid point set, `dltCalibrateCameras()` uses `nlminb()` to search for the six transformation parameters per grid that minimizes the RMS error when the 3D coordinates are input (with the corresponding 2D coordinates) to `dltCoefficients`. In effect, `dltCalibrateCameras()` solves for the position of each grid in 3D space using the error from `dltCoefficients` as an optimality criterion. Since the first grid is fixed, the optimization will search for 6*(n-1) parameters, where n is the number of separate grid orientations. `nlminb()` calls the function `dltTransformationParameterRMSError`.

In order to fully explore the parameter space, `dltCalibrateCameras()` calls `nlminb()` several times with a different set of randomly generated starting parameters to estimate the transformation parameters for each additional grid. The number of different sets of starting parameters is determined by `nlm.calls.max`. An initial optimization run is intended to quickly determine whether a particular set of starting parameters is likely to lead to convergence. The number of iterations for

this initial optimization is determined by `nlm.iter.max.init` and the objective used in determining likely convergence is `objective.min.init`. If it is determined that the starting parameters are likely to lead to convergence below `objective.min`, `nlminb()` is allowed to continue optimizing. For each grid, the solution yielding the lowest error, or the first solution below the `objective.min` threshold, is retained for the next grid optimization.

These optimal transformation parameters are then used to obtain the 3D coordinates of the original grid points (not downsampled). Once these 3D coordinates are known, the 3D and 2D pixel coordinates are input to `dltCoefficients` to obtain the 11 calibration coefficients per camera. For this reason, the calibration coefficient RMS Error (`coefficient.rmse`) returned will differ slightly from the reported final `nlminb()` minimum (`t.min`).

If `c.run` is set to `TRUE`, `dltCalibrateCameras()` performs a second optimization on the calibration coefficients themselves. `nlminb()` is used, this time calling `dltCoefficientRMSError`, to find the 11 calibration coefficients per view that minimizes the reconstruction RMS error. Note that `dltCoefficients` cannot be used as with the previous optimization because the coefficients must be an input. Running this second optimization seems to have little effect in increasing the accuracy of the calibration but is included as this may be useful for some stereo setups.

For a step-by-step tutorial on how to use `dltCalibrateCameras()` see Calibrating stereo cameras.

Value

a list of class `"dltCalibrateCameras"` with the following elements:

`cal.coeff`	a matrix of 11 optimized DLT calibration coefficients per camera view.
`coor.3d`	the optimized 3D coordinates of the input grid points in `coor.2d`.
`mean.reconstruct.rmse`	the RMS error when `coor.2d` and the optimized calibration coefficients `cal.coeff` are input to `dltReconstruct`.
`coefficient.rmse`	the RMS error when `coor.2d` and the optimized 3D coordinates `coor.3d` are input to `dltCoefficients`.
`t.param.final`	the final transformation parameters reported by `nlminb()` from the first optimization. `'t.'` refers to the transformation optimization.
`t.min`	the minimum reported by `nlminb()` from the first optimization. This is the mean RMS error across all camera views returned by `dltCoefficients` for the downsampled grid points.
`t.runtime`	the run-time (in seconds) for the first optimization.

if `c.run` is `FALSE`, the following are `NA`. Otherwise,

`c.param.init`	the initial parameters for the second optimization. `'c.'` refers to the coefficient optimization.
`c.param.final`	the final parameters reported by `nlminb()` from the second optimization.
`c.min`	the minimum reported by `nlminb()` from the second optimization. This is the mean RMS error across all camera views returned by `dltReconstruct`.
`c.iter`	the number of iterations reported by `nlminb()` from the second optimization.
`c.runtime`	the run-time (in seconds) for the second optimization.

Author(s)

Aaron Olsen

References

For a general overview of DLT: http://kwon3d.com/theory/dlt/dlt.html

See Also

dltTestCalibration, dltCoefficients, readCheckerboardsToArray, transformPlanarCalibrationCoordinates, dltTransformationParameterRMSError, dltCoefficientRMSError

Examples

```
## SET NUMBER OF INTERNAL CORNERS FOR CALIBRATION GRIDS
nx <- 21
ny <- 14

## GET THE FILE DIRECTORY FOR EXTRA R PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## SET FILE PATH TO CHECKERBOARD CORNERS FROM CALIBRATION IMAGE SET
## THE TUTORIAL INCLUDES 8 CALIBRATION IMAGES FROM TWO CAMERA VIEWS
file <- matrix(c(paste0(fdir, "cal_a", 1:8, "_v1.txt"),
  paste0(fdir, "cal_a", 1:8, "_v2.txt")), ncol=2)

## READ IN CHECKERBOARD CORNERS
## NOTE THAT col.reverse IS USED TO MAKE POINTS CORRESPOND
coor.2d <- readCheckerboardsToArray(file=file, nx=nx, ny=ny, col.reverse=TRUE)

## SET GRID SIZE (IN MM)
grid.size <- 6.347889

## Not run:
## CALIBRATE CAMERAS
## TO REDUCE RUN-TIME, WE JUST USE CORNERS FROM TWO IMAGES (1 AND 5)
dlt_calibrate_cameras <- dltCalibrateCameras(coor.2d=coor.2d[, , c(1, 5), ], nx=nx,
  grid.size=grid.size, c.run=FALSE, print.progress=TRUE)

## RUN CALIBRATION ON ALL IMAGES, ACCURACY IS GREATLY IMPROVED
dlt_calibrate_cameras <- dltCalibrateCameras(coor.2d=coor.2d, nx=nx,
  grid.size=grid.size, c.run=FALSE, print.progress=TRUE)

## PRINT SUMMARY
summary(dlt_calibrate_cameras)

## End(Not run)
```

dltCoefficientRMSError

Returns the error during calibration coefficient optimization

Description

Returns the RMS error from `dltReconstruct` when optimizing the calibration coefficients. This function is called internally by `dltCalibrateCameras`.

Usage

```
dltCoefficientRMSError(p, coor.2d)
```

Arguments

p	a vector of the current, 11-parameter calibration coefficients.
coor.2d	a four-dimensional array of grid points passed from `dltCalibrateCameras`.

Value

the mean RMS error from `dltReconstruct` across all views.

Author(s)

Aaron Olsen

See Also

`transformPlanarCalibrationCoordinates`, `dltReconstruct`, `dltCalibrateCameras`

dltCoefficients

Computes DLT coefficients for a stereo camera setup

Description

This function takes 3D coordinates and their corresponding 2D coordinates in one or more camera views and returns DLT calibration coefficients. The DLT coefficients can then be used in 3D reconstruction and calculation of epipolar lines.

Usage

```
dltCoefficients(coor.3d, coor.2d)
```

Arguments

coor.3d	a three-column matrix of 3D coordinates.
coor.2d	an three-dimensional array of 2D pixel coordinates.

Details

This function takes 3D coordinates and their corresponding 2D coordinates in one or more camera views and returns DLT calibration coefficients. Note that to find the calibration coefficient for a particular camera view, only the pixel coordinates in that camera view and their corresponding 3D coordinates are used. Thus, it is possible to derive calibration coefficients for several cameras without any overlap among the views in the points used to derive the calibration coefficients. Any missing values (either in coor.3d or pixel coordinates missing in a particular view in coor.2d) can be input as NA; they will be ignored.

The requirements for the structure of the coor.2d array are as follows. The first dimension of coor.2d is the number of points used in calculating the DLT coefficients. The number of elements in the first dimension of coor.2d must match the number of rows in coor.3d and these must be corresponding points (though some may be NA). The second dimension of coor.2d should be two as these are x,y-coordinates. The third dimension of coor.2d is the number of camera views. This will correspond to the number of columns in the returned calibration coefficient matrix.

For more information on stereo camera calibration, see Calibrating stereo cameras.

Value

a list of class "dltCoefficients" with the following elements:

cal.coeff	the calibration coefficient matrix.
rmse	the root-mean-square error for each camera view.

Note

This function was modified by A Olsen from the Matlab function dlt_reconstruct() written by T Hedrick.

Author(s)

Aaron Olsen

References

Abdel-Aziz, Y.I., Karara, H.M. (1971) Direct linear transformation into object space coordinates in close-range photogrammetry. *Proc. Symp. on Close-Range Photogrammetry* (University of Illinois at Urbana-Champaign).

Hedrick, T.L. (2008) Software techniques for two- and three-dimensional kinematic measurements of biological and biomimetic systems. *Bioinspiration & Biomimetics*, **3** (034001).

For a general overview of DLT: http://kwon3d.com/theory/dlt/dlt.html

See Also

dltCalibrateCameras, findCheckerboardCorners

Examples

```
## SET NUMBER OF INTERNAL CORNERS AND ROWS
nx <- 21
ny <- 14

## GET THE FILE DIRECTORY FOR EXTRA R PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## GET FILE PATHS FOR PIXEL COORDINATES FROM CALIBRATION IMAGES 1 AND 5
file2d <- matrix(c(paste0(fdir, "cal_a", c(1, 5), "_v1.txt"),
  paste0(fdir, "cal_a", c(1, 5), "_v2.txt")), ncol=2)

## READ INTO ARRAY
## THESE ARE THE 2D COORDINATES USED IN THE CALIBRATION
coor.2d <- readCheckerboardsToArray(file=file2d, nx=nx, ny=ny, col.reverse=TRUE)

## REDUCE ARRAY DIMENSIONS TO THREE
## THIS STACKS MULTIPLE VIEWS ON TOP OF EACH OTHER INTO THE SAME MATRIX
coor.2d <- apply(coor.2d, c(2, 4), matrix, byrow=FALSE)

## GET FILE PATH FOR CORRESPONDING 3D COORDINATES
file3d <- paste0(fdir, "cal_3d_coor.txt")

## READ 3D POINTS INTO MATRIX
coor.3d <- as.matrix(read.table(file=file3d))

## FIND THE DLT COEFFICIENTS
dlt_coeffcients <- dltCoefficients(coor.3d=coor.3d, coor.2d=coor.2d)

## PRINT THE SUMMARY
summary(dlt_coeffcients)

## NOTE THAT EACH CAMERA VIEW IS CALIBRATED SEPARATELY
## GIVES THE EXACT SAME RESULT
dltCoefficients(coor.3d=coor.3d, coor.2d=coor.2d[, , 1])
```

---

dltEpipolarDistance *Finds the distance between a point and a self-epipolar line*

Description

Given the same point in two camera views, this function finds the distance between the point in the second camera view and a epipolar line in the second view, as determined from the point in the first view. The option is also available to return the mean reciprocal epipolar distance.

Usage

```
dltEpipolarDistance(p1, p2, cal.coeff, reciprocal = FALSE)
```

Arguments

p1	an x,y vector or two-column matrix of a point or points in the camera view corresponding to the first column of cal.coeff. This point will be used to generate an epipolar line in the second view.
p2	an x,y vector or two-column matrix of a point or points in a second camera view, corresponding to the second column of cal.coeff. The distance will be measured from this point to the epipolar line of p1.
cal.coeff	a two-column matrix of DLT calibration coefficients, where each column corresponds to the views from which p1 and p2 were taken, respectively.
reciprocal	a logical indicating whether epipolar distance should be calculated reciprocally and then averaged.

Details

In a stereo camera setup, a point in one camera view must fall somewhere along a line in a second camera view. This line is called its epipolar line. Due to error in manually selecting the same point in two camera views and error in the calibration, the epipolar line of the point in the first view will not intersect exactly with the same point in the second view. This distance between a point and the epipolar line of the same point in another view is the epipolar distance (or error).

The epipolar distance can be calculated between the point in the second view and the epipolar line of the point in the first view or between the point in the first view and the epipolar line of the point in the second view; the choice is arbitrary. This function performs the former. If a user would like to perform the latter, simply switch p1 with p2 *and* reverse the column order of cal.coeff (see "Examples"). Another possibility is to perform both distance calculations and return an average (mean reciprocal epipolar distance). This can be done by setting reciprocal to TRUE.

Although a stereo camera system may consist of more than two cameras, the coefficients of only two cameras should be input to dltEpipolarDistance(). Only the coefficients of the two camera views for which epipolar distances are being calculated are relevant. Currently, this function only works with the 11-parameter DLT model.

A few options for input of p1 and p2 are available. If a single point is input for both, the epipolar distance is calculated for these two points. If a matrix of points is input for both (of the same dimensions), the epipolar distance is calculated pair-wise - points in the same row are treated as the same point. Lastly, if a single point is input as p1 and a matrix is input as p2, the epipolar distance is calculated for p1 relative to all points in p2 (see "Examples").

Value

a vector of the epipolar distance(s).

Author(s)

Aaron Olsen

References

For a general overview of DLT: http://kwon3d.com/theory/dlt/dlt.html

See Also

dltCalibrateCameras, dltEpipolarLine, dltNearestPointOnEpipolar,

dltMatchCurvePoints

Examples

```
## FIND THE FILE DIRECTORY FOR EXTRA R PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## SET FILE PATH TO CALIBRATION COEFFICIENTS IN TWO CAMERA STEREO SETUP
cc_file <- paste0(fdir, "cal_coeffs.txt")

## LOAD COEFFICIENTS
cal.coeff <- as.matrix(read.table(file=cc_file))

## GET LANDMARKS IN FIRST CAMERA VIEW
lm_files <- paste0(fdir, c("lm_2d_a1_v1.txt", "lm_2d_a1_v2.txt"))

## READ LANDMARKS INTO MATRIX
lm.array <- readLandmarksToArray(file=lm_files, row.names=1)

## FIND EPIPOLAR DISTANCE BETWEEN TWO SINGLE LANDMARKS
## EPIPOLAR DISTANCE (ERROR) IS AROUND 7 PIXELS
## IDENTIFYING THE EXACT SAME POINT IN TWO VIEWS MANUALLY IS CHALLENGING...
dltEpipolarDistance(p1=lm.array[1, , 1], p2=lm.array[1, , 2], cal.coeff=cal.coeff)

## FIND EPIPOLAR DISTANCE USING EPIPOLAR FROM SECOND VIEW INSTEAD
dltEpipolarDistance(p1=lm.array[1, , 2], p2=lm.array[1, , 1], cal.coeff=cal.coeff[, 2:1])

## FIND MEAN RECIPROCAL EPIPOLAR DISTANCE BETWEEN TWO SINGLE LANDMARKS
## THIS IS THE AVERAGE OF THE PREVIOUS TWO DISTANCES
dltEpipolarDistance(p1=lm.array[1, , 1], p2=lm.array[1, , 2], cal.coeff=cal.coeff,
  reciprocal=TRUE)

## FIND EPIPOLAR DISTANCES BETWEEN ALL LANDMARKS
## PROCEEDS PAIRWISE BECAUSE p1 AND p2 HAVE THE SAME DIMENSIONS
dltEpipolarDistance(p1=lm.array[, , 1], p2=lm.array[, , 2], cal.coeff=cal.coeff)

## FIND EPIPOLAR DISTANCES BETWEEN FIRST LANDMARK AND ALL LANDMARKS
## HERE THE EPIPOLAR DISTANCES ARE HIGH BECAUSE ONLY THE FIRST LANDMARK
##  CORRESPONDS
## THE REMAINING POINTS ARE NOT THE SAME LANDMARK
dltEpipolarDistance(p1=lm.array[1, , 1], p2=lm.array[, , 2], cal.coeff=cal.coeff)
```

`dltEpipolarLine` *Finds a epipolar or self-epipolar line*

Description

This function takes a point in one camera view and returns either its epipolar line in another camera view or its epipolar line in that same camera view (self-epipolar line).

Usage

```
dltEpipolarLine(p, cal.coeff1, cal.coeff2 = NULL, self = FALSE)
```

Arguments

`p`	vector of x,y pixel coordinates for a point in an image.
`cal.coeff1`	DLT calibration coefficients corresponding to the camera view from which `p` is taken or a two-column matrix of calibration coefficients in which the first column corresonds to the camera view from which `p` is taken and the second column corresponds to an additional camera view.
`cal.coeff2`	in the case that `cal.coeff1` is a single column matrix, these are the DLT calibration coefficients corresponding to a camera view in a stereo camera setup other than that from which `p` is taken.
`self`	a logical indicating whether the epipolar line returned should be a self-epipolar line.

Details

In a stereo camera setup, a point in one camera view must fall somewhere along a line in a second camera view. This line is called its epipolar line. If a second point is taken anywhere along this epipolar line in the second camera and its epipolar line is found in the first camera, the original point must fall along this line. The epipolar line in the first camera view, along which the original point falls, is called its self-epipolar line (Yakutieli et al. 2007). `dltEpipolarLine()` uses DLT calibration coefficients (see `dltCalibrateCameras`) to find the epipolar or self-epipolar line for a given point in a stereo camera setup.

Although a stereo camera system may consist of more than two cameras, the coefficients of only two cameras should be input to `dltEpipolarLine()`. Only the coefficients of the two cameras between which epipolar lines are being calculated are relevant. These two columns of coefficients can be input as one matrix (to `cal.coeff1`) or as two separate, one-column matrices (to `cal.coeff1` and `cal.coeff2`).

Currently, `dltEpipolarLine()` only works with the 11-parameter DLT model.

Value

`dltEpipolarLine()` outputs the resulting epipolar line in two forms: slope-intercept coefficients (`m` and `b`) and two points on the line (`l1` and `l2`). These are stored in a list as follows:

m	the slope of the epipolar line.
b	the y-intercept of the epipolar line.
l1	one point on the epipolar line.
l2	a second point on the epipolar line.

Note

This function was modified by A Olsen from the Matlab function `partialdlt()` written by T Hedrick. A Olsen added the self-epipolar functionality after Yekutieli et al. 2007.

Author(s)

Aaron Olsen

References

Abdel-Aziz, Y.I., Karara, H.M. (1971) Direct linear transformation into object space coordinates in close-range photogrammetry. *Proc. Symp. on Close-Range Photogrammetry* (University of Illinois at Urbana-Champaign).

Yekutieli, Y., Mitelman, R., Hochner, B. & Flash, T. (2007). Analyzing Octopus Movements Using Three-Dimensional Reconstruction. *Journal of Neurophysiology*, **98**, 1775–1790.

Hedrick, T.L. (2008) Software techniques for two- and three-dimensional kinematic measurements of biological and biomimetic systems. *Bioinspiration & Biomimetics*, **3** (034001).

For a general overview of DLT: http://kwon3d.com/theory/dlt/dlt.html

See Also

dltCalibrateCameras, dltEpipolarDistance, dltNearestPointOnEpipolar

Examples

```
## FIND THE FILE DIRECTORY FOR EXTRA R PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## SET FILE PATH TO CALIBRATION COEFFICIENTS IN TWO CAMERA STEREO SETUP
cc_file <- paste0(fdir, "cal_coeffs.txt")

## LOAD COEFFICIENTS
cal.coeff <- as.matrix(read.table(file=cc_file))

## GET LANDMARKS IN FIRST CAMERA VIEW
lm_file <- paste0(fdir, "lm_2d_a1_v1.txt")

## READ LANDMARKS INTO MATRIX
lm.matrix <- readLandmarksToMatrix(file=lm_file, row.names=1)

## FIND EPIPOLAR LINE IN SECOND CAMERA VIEW
epipolar <- dltEpipolarLine(p=lm.matrix['occipital_condyle', ], cal.coeff1=cal.coeff)
```

```
## FIND SELF-EPIPOLAR LINE (IN FIRST CAMERA VIEW)
self_epipolar <- dltEpipolarLine(p=lm.matrix['occipital_condyle', ], cal.coeff1=cal.coeff,
  self=TRUE)

## CONFIRM THAT DISTANCE FROM ORIGINAL POINT TO SELF-EPIPOLAR LINE IS ZERO
distancePointToLine(p=lm.matrix['occipital_condyle', ], l1=self_epipolar)
```

`dltInverse` *Returns ideal pixel coordinates of 3D point(s) in a stereo camera setup*

Description

This function takes 3D coordinates and the DLT calibration coefficients corresponding to one camera view and returns the ideal pixel coordinates of the 3D points in that camera view.

Usage

```
dltInverse(cal.coeff, coor.3d)
```

Arguments

`cal.coeff`	a single column matrix of DLT calibration coefficients for one camera view.
`coor.3d`	a three-column matrix of 3D coordinates.

Details

When `dltReconstruct` is used to reconstruct points in 3D based on pixel coordinates from two or more camera views, these 3D points can be projected back into any camera view at their "ideal" pixel coordinates (the "inverse" of reconstruction). The "ideal" pixel coordinates represent the pixel coordinates in each view if there were no error (i.e. all pixel coordinates in every view correspond to the exact same point in 3D). In any real-world system there is some error and these ideal pixel coordinates are compared to the original pixel coordinates used in the reconstruction to assess reconstruction error. `dltInverse()` is called by `dltCoefficients` and `dltEpipolarLine`.

Since `dltInverse()` only projects the 3D coordinates into a single camera view, only one column of the DLT coefficients should be input. Currently, `dltInverse()` only works with the 11-parameter DLT model.

Value

a two-column matrix of pixel coordinates of all points in `coor.3d` in the camera view corresponding to `cal.coeff`.

Note

This function was modified by A Olsen from the Matlab function `dlt_inverse()` written by T Hedrick.

Author(s)

Aaron Olsen

References

Abdel-Aziz, Y.I., Karara, H.M. (1971) Direct linear transformation into object space coordinates in close-range photogrammetry. *Proc. Symp. on Close-Range Photogrammetry* (University of Illinois at Urbana-Champaign).

Hedrick, T.L. (2008) Software techniques for two- and three-dimensional kinematic measurements of biological and biomimetic systems. *Bioinspiration & Biomimetics*, **3** (034001).

For a general overview of DLT: http://kwon3d.com/theory/dlt/dlt.html

See Also

dltCalibrateCameras, dltReconstruct

Examples

```
## GET THE FILE DIRECTORY FOR EXTRA R PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## SET FILE PATH TO CALIBRATION COEFFICIENTS IN TWO CAMERA STEREO SETUP
cc_file <- paste0(fdir, "cal_coeffs.txt")

## LOAD COEFFICIENTS
cal.coeff <- as.matrix(read.table(file=cc_file))

## READ LANDMARKS INTO MATRIX
lm.matrix <- readLandmarksToMatrix(file=paste0(fdir, "lm_3d_a1.txt"), row.names=1)

## GET IDEAL 2D COORDINATES OF 3D POINTS IN FIRST CAMERA VIEW
dltInverse(cal.coeff[, 1], lm.matrix)

## GET IDEAL 2D COORDINATES OF 3D POINTS IN SECOND CAMERA VIEW
dltInverse(cal.coeff[, 2], lm.matrix)
```

---

dltMatchCurvePoints *Matches curve points between two camera views*

Description

This function uses DLT calibration coefficients to find corresponding points along a curve viewed from two different cameras in stereo camera setup.

Usage

```
dltMatchCurvePoints(lm.list, cal.coeff, ref.view = "max", window.size = 30,
                    min.tangency.angle = 0.1, min.dist.adj.slope = 0.1,
                    plot.match = FALSE, fill.missing=TRUE)

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

Arguments

`lm.list`	a list of curve points from two camera views (see `readLandmarksToList`). `lm.list` can include landmarks; these will be returned unchanged.
`cal.coeff`	a two-column matrix of DLT calibration coefficients, where each column corresponds to the views from which points in `lm.list` were taken.
`ref.view`	which of the two views to use as a reference. Possible values are `1`, `2`, `"max"` and `"min"`.
`window.size`	the number of curve points to consider in each corresponding point search.
`min.tangency.angle`	a threshold for which parts of the curve should not be matched because they are nearly parallel to the epipolar line. See "Details".
`min.dist.adj.slope`	a threshold for which parts of the curve should not be matched because a point is not clearly the best match in comparison with neighboring points.
`plot.match`	a logical indicating whether a plot should be drawn showing the point matches between two curves. This is not optimized for visualization and might not always provide the best view of the matching results.
`fill.missing`	a logical whether missing segments should be filled.
`object`	a list of class `"dltMatchCurvePoints"` (the output of `dltMatchCurvePoints()`).
`...`	further arguments passed to or from other methods.

Details

Point reconstruction with a stereo camera setup requires pixel coordinates of the same point in two or more camera views. Curves can also be reconstructed in a stereo camera setup if reconstructed as a series of single points. This, however, poses an additional challenge: that of identifying the same point on a curve viewed from two different perspectives. A point half-way along the curve in one view will not necessarily correspond to a point half-way along the same curve in another view. `dltMatchCurvePoints()` solves this challenge by using epipolar geometry informed by the DLT calibration coefficients (Yekutieli et al. 2007).

In a stereo camera setup, a point in one camera view must fall somewhere along a line in a second camera view. This line is called its epipolar line. If the same curve has been digitized in two camera views, the epipolar line of a point on the first curve should intersect the curve in the second camera view. The point at which the epipolar line intersects the curve in the second view represents the corresponding point on the second curve. `dltMatchCurvePoints()` iterates through all the points on one curve and uses epipolar geometry to identify the corresponding point on a second curve. The corresponding point is identified as a point on the epipolar line that is closest to the curve in the

second view (rather than finding the intersection, per se). For more details on the use of epipolar geometry to solve for corresponding points see Yekutieli et al. (2007).

Two different types of curve point input to `dltMatchCurvePoints()` are possible. The first type is a list with two elements (`list[[1]]` and `list[[2]]`), containing the curve points of the first and second camera view, respectively. The second type is a list of the same form as the landmark list described in `readLandmarksToList`. The main elements of the landmark list are the landmarks and curves (`list[['landmark1']]`, `list[['curve1']]`, etc.). Each main element then has two elements (e.g. `list[['curve1']][[1]]`, `list[['curve1']][[2]]`) corresponding to the first and second camera views, respectively. The curve points themselves should be densely sampled pixel coordinates (e.g. single pixel spacing) in order to improve matching accuracy.

`dltMatchCurvePoints()` returns the landmark list as the element `match.lm.list` in the same format as the input, except that all curve points will be corresponding points. Note that list input is used, rather than a matrix, because the number of curve points may differ between the two views. Once the corresponding curve points are identified, however, the number of curve points in each view will be equal. Landmarks and curves containing less than three points are ignored and returned just as input. In this way, all landmarks and curve points can be passed through `dltMatchCurvePoints()` without having to be processed separately.

Although a stereo camera system may consist of more than two cameras, the coefficients of only two cameras should be input to `dltMatchCurvePoints()`. Only the coefficients of the two camera views for which corresponding curve points are being identified are relevant. Currently, this function can only match curve points between two camera views using the 11-parameter DLT model.

The curve points chosen as the reference are used to generate epipolar lines in a second camera view. The results will differ slightly depending on which view is chosen as a reference. By default, `dltMatchCurvePoints()` uses the curve with the maximum number of points as a reference. Users can specify which view is to be used as reference through `ref.view`. Setting `ref.view` to "min" will use the curve with the minimum number of points as a reference. Setting `ref.view` to 1 or 2 will use the first view or second view as a reference, respectively.

As `dltMatchCurvePoints()` steps through each point on the reference curve, it searches for the closest point on the epipolar line to the second curve. Rather than search for the closest point among all of the second curve points, `dltMatchCurvePoints()` only searches over a sliding window of points. `window.size` is the number of curve points considered at each iteration in identifying the corresponding point. A lower `window.size` will decrease the run-time but will potentially cause the function to miss corresponding points. If `curve.pt.dist` values are low, the current `window.size` is probably appropriate. `window.size` can be increased if `curve.pt.dist` values are high (over several pixels).

When the epipolar line is nearly parallel to the curve in the non-reference view, several points are equally likely to be the corresponding point and determining the actual corresponding point is impossible without additional information. The angle between the epipolar line and the points on the non-reference curve is referred to here as the tangency angle. When the tangency angle for points on the non-reference curve is less than `min.tangency.angle`, the current reference point is skipped. Additionally, when the points near a point on the non-reference curve are also very close to the epipolar line, a wrong match is more likely. Within the window of candidate points, the distance from each point to the epipolar line is calculated. The slope of these distances away from the point closest to the epipolar line (the minimum distance) is referred to here as the adjacent point distance slope. When the adjacent point distance slope is lower, the confidence that the minimum distance point is the correct match decreases. When this adjacent point distance slope is less than `min.dist.adj.slope`, the current reference point is skipped. The `min.tangency.angle` and

min.dist.adj.slope are similar criteria, however the min.dist.adj.slope might provide more robust results with more irregular curves. Users might need to increase one or both of these values to obtain satisfactory results.

When reference points are skipped, these are filled in at the end with straight lines extending between defined points to either side of the skipped regions. Straight lines are used because these regions are likely to be nearly linear, owing to their minimal deviation from the epipolar line.

In addition to returning match.lm.list, dltMatchCurvePoints() also returns two vectors (or lists of vectors, depending on the format of lm.list) that can be used to assess the accuracy of the curve point matching. epipolar.dist is the epipolar distance between the epipolar line of the reference point and the corresponding point in the non-reference view. The first and last point are assumed to correspond, so there will be some epipolar error for these points. The remaining points are chosen on the epipolar line of the reference point, so their epipolar error will be zero. Future implementations may allow users to specify that corresponding points be on the curve in the second view and not necessarily on the epipolar line, in which case epipolar.dist will become more relevant. curve.pt.dist is the distance between the epipolar line and the nearest point on the curve in the second view for each curve point. If the exact same curve has been digitized in the two views, curve.pt.dist should be low (within a pixel or less).

For a tutorial on how to use dltMatchCurvePoints() see Reconstructing 2D points and curves into 3D.

Value

a list of class "dltMatchCurvePoints" with the following elements:

match.lm.list	a landmark list of matched curve points (and landmarks if also input).
epipolar.dist	a list or vector of the epipolar distance between the epipolar line of the reference points and the corresponding non-reference point. In current implementation, all values will be zero except the start and end points.
curve.pt.dist	a list or vector of the distances from the chosen corresponding points and the nearest point on the non-reference curve.

Note

This function was written by A Olsen based on the methodology described in Yekutieli et al. 2007.

Author(s)

Aaron Olsen

References

Yekutieli, Y., Mitelman, R., Hochner, B. and Flash, T. (2007). Analyzing Octopus Movements Using Three-Dimensional Reconstruction. *Journal of Neurophysiology*, **98**, 1775–1790.

For a general overview of DLT: http://kwon3d.com/theory/dlt/dlt.html

See Also

readLandmarksToList, dltEpipolarLine, dltEpipolarDistance, dltNearestPointOnEpipolar

Examples

```
## GET THE FILE DIRECTORY FOR EXTRA R PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## SET FILE PATH TO LANDMARK DATA
file <- paste0(fdir, "lm_2d_a2_v", 1:2, ".txt")

## LOAD COEFFICIENTS
cal.coeff <- as.matrix(read.table(file=paste0(fdir, "cal_coeffs.txt")))

## READ LANDMARKS INTO LIST
lm.list <- readLandmarksToList(file=file, row.names=1)

## MATCH CURVE POINTS FOR ONE CURVE
## FIRST TYPE OF LANDMARK INPUT
## RETURNS LIST OF MATCHING POINTS WITHOUT CURVE NAME
dlt_match <- dltMatchCurvePoints(lm.list$pterygoid_crest_R, cal.coeff)

## PRINT SUMMARY
summary(dlt_match)

## SET A DIFFERENT REFERENCE VIEW
## SECOND VIEW HAS 80 FEWER POINTS
## dlt_match <- dltMatchCurvePoints(lm.list$pterygoid_crest_R, cal.coeff, ref.view=2)

## MATCH CURVE POINTS FOR ALL CURVES IN LIST
## SECOND TYPE OF LANDMARK INPUT
## RETURNS LIST OF ALL LANDMARKS AND MATCHED CURVE POINTS WITH CURVE NAMES
## dlt_match <- dltMatchCurvePoints(lm.list, cal.coeff)
```

---

dltNearestPointOnEpipolar

Returns the closest point on a epipolar line to a point or points

---

Description

Given the same point in two camera views, this function finds the nearest point on the epipolar line of the point in the first view to a point or points in the second view.

Usage

```
dltNearestPointOnEpipolar(p1, p2, cal.coeff)
```

Arguments

p1	vector of x,y pixel coordinates for a point in the camera view corresponding to the first column of `cal.coeff`. This point will be used to generate an epipolar line in the second view.

p2	an x,y vector or two-column matrix of a point or points in a second camera view, corresponding to the second column of cal.coeff. The nearest point on the epipolar line will be an orthogonal projection from a point in p2.
cal.coeff	a two-column matrix of DLT calibration coefficients, where each column corresponds to the views from which p1 and p2 were taken, respectively.

Details

In a stereo camera setup, a point in one camera view must fall somewhere along a line in a second camera view. This line is called its epipolar line. Due to error in manually selecting the same point in two camera views and error in the calibration, the epipolar line of the point in the first view will not intersect exactly with the same point in the second view. The nearest point on the epipolar line is a point at a minimum distance from the point in the second view. This is equivalent to the orthogonal projection (orthogonalProjectionToLine) of the point in the second view onto the epipolar line of the point in the first view. The length of this line is the epipolar distance (dltEpipolarDistance).

dltNearestPointOnEpipolar() first finds the epipolar line of p1, a point in the first camera view, and then finds the point on this epipolar line nearest to point(s) p2 in the second camera view. If p2 is a single point, dltNearestPointOnEpipolar() finds the point on the epipolar line closest to p2. If p2 is a matrix of points, the point in p2 closest to the epipolar line is first identified and then the point on the epipolar line closest to this point is determined.

Value

a list with the following elements:

matching.pt	an x,y vector of the point on the epipolar line of p1 closest to point(s) p2.
min.idx	the index in p2 of the nearest point to the epipolar line of p1. If p2 is a single point (vector), min.idx will be 1.
p2.dist	the epipolar distance between matching.pt and the point in p2 at the min.idx.

Author(s)

Aaron Olsen

References

For a general overview of DLT: http://kwon3d.com/theory/dlt/dlt.html

See Also

dltCalibrateCameras, dltEpipolarDistance, dltEpipolarLine, dltMatchCurvePoints

Examples

```
## FIND THE FILE DIRECTORY FOR EXTRA R PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## SET FILE PATH TO CALIBRATION COEFFICIENTS IN TWO CAMERA STEREO SETUP
```

```
cc_file <- paste0(fdir, "cal_coeffs.txt")

## LOAD COEFFICIENTS
cal.coeff <- as.matrix(read.table(file=cc_file))

## GET LANDMARKS IN FIRST CAMERA VIEW
lm_files <- paste0(fdir, c("lm_2d_a1_v1.txt", "lm_2d_a1_v2.txt"))

## READ LANDMARKS INTO MATRIX
lm.array <- readLandmarksToArray(file=lm_files, row.names=1)

## FIND THE NEAREST POINT ON THE EPIPOLAR LINE OF P1
dltNearestPointOnEpipolar(p1=lm.array[3, , 1], p2=lm.array[3, , 2], cal.coeff=cal.coeff)

## FIND THE NEAREST POINT ON THE EPIPOLAR LINE OF P1
## THIS TIME USING ALL LANDMARKS IN THAT VIEW
## FUNCTION IDENTIFIES THE CORRECT LANDMARK IN THE SECOND VIEW AS THE SAME LANDMARK
dltNearestPointOnEpipolar(p1=lm.array[3, , 1], p2=lm.array[, , 2], cal.coeff=cal.coeff)
```

`dltReconstruct` *Reconstructs the 3D position of points in two or more camera views*

Description

This function takes 2D pixel coordinates of a point or points from two more camera views and uses DLT coefficients to reconstruct their position in 3D.

Usage

```
dltReconstruct(cal.coeff, coor.2d, min.views = 2)

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

Arguments

`cal.coeff`	a matrix of DLT calibration coefficients. The columns correspond to each camera view and the column order should match the camera view order of the landmarks in `coor.2d`.
`coor.2d`	2D pixel coordinates from two or more camera views. Format can be either a landmark matrix, list or array.
`min.views`	the minimum number of views required for a point to be reconstructed in 3D.
`object`	a list of class `"dltReconstruct"` (the output of `dltReconstruct()`).
`...`	further arguments passed to or from other methods.

Details

This function uses DLT coefficients (calculated using dltCalibrateCameras, for example) to reconstruct the 3D position of points, based on their 2D position in two or more camera views. 2D pixel coordinates can be input as a landmark matrix (readLandmarksToMatrix), as a list (readLandmarksToList) or as an array (readLandmarksToArray).

A minimum of two views is required for 3D reconstruction although additional camera views can be used, potentially improving reconstruction accuracy. Points that are present in fewer views than specified by `min.views` will be assigned NA values in the returned 3D matrix (`coor.3d`).

After 3D reconstruction, `dltReconstruct()` performs the inverse operation, taking the reconstructed, 3D coordinates and solving for the 2D position of the points in each camera view. These inverse 2D coordinates are compared with the original coordinates and their difference is returned as the root-mean-square (RMS) reconstruction error (`list$rmse`). This error is similar to the epipolar distance (dltEpipolarDistance). The `summary()` function can be used to view the error by landmark.

Currently, `dltReconstruct()` only works with the 11-parameter DLT model.

For a step-by-step tutorial on how to use `dltReconstruct()` see Reconstructing 2D points and curves into 3D.

Value

a list of class `"dltReconstruct"` with the following elements:

`coor.3d`	a 2D or 3D landmark matrix.
`rmse`	the root-mean-square reconstruction error (in pixels).

Note

This function was modified by A Olsen from the Matlab function `dlt_reconstruct()` written by T Hedrick.

Author(s)

Aaron Olsen

References

Abdel-Aziz, Y.I., Karara, H.M. (1971) Direct linear transformation into object space coordinates in close-range photogrammetry. *Proc. Symp. on Close-Range Photogrammetry* (University of Illinois at Urbana-Champaign).

Hedrick, T.L. (2008) Software techniques for two- and three-dimensional kinematic measurements of biological and biomimetic systems. *Bioinspiration & Biomimetics*, **3** (034001).

For a general overview of DLT: http://kwon3d.com/theory/dlt/dlt.html

See Also

dltCalibrateCameras, readLandmarksToMatrix, readLandmarksToList, readLandmarksToArray, dltEpipolarDistance

Examples

```
## GET THE FILE DIRECTORY FOR EXTRA R PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## SET FILE PATH TO CALIBRATION COEFFICIENTS IN TWO CAMERA STEREO SETUP
cc_file <- paste0(fdir, "cal_coeffs.txt")

## LOAD COEFFICIENTS
cal.coeff <- as.matrix(read.table(file=cc_file))

## GET LANDMARKS IN FIRST CAMERA VIEW
lm_files <- paste0(fdir, c("lm_2d_a1_v1.txt", "lm_2d_a1_v2.txt"))

## READ LANDMARKS INTO MATRIX
lm.matrix <- readLandmarksToMatrix(file=lm_files, row.names=1)

## RECONSTRUCT LANDMARKS IN 3D (MATRIX INPUT)
dlt_recon <- dltReconstruct(cal.coeff=cal.coeff, coor.2d=lm.matrix)


## OTHER POSSIBLE LANDMARK FORMAT INPUTS ##
## READ LANDMARKS INTO LIST
lm.list <- readLandmarksToList(file=lm_files, row.names=1)

## RECONSTRUCT LANDMARKS IN 3D (LIST INPUT)
dlt_recon <- dltReconstruct(cal.coeff=cal.coeff, coor.2d=lm.list)

## READ LANDMARKS INTO ARRAY
lm.array <- readLandmarksToArray(file=lm_files, row.names=1)

## RECONSTRUCT LANDMARKS IN 3D (ARRAY INPUT)
dlt_recon <- dltReconstruct(cal.coeff=cal.coeff, coor.2d=lm.array)
```

---

dltTestCalibration *Tests the accuracy of a stereo camera calibration*

Description

This function uses a set of grid points, ideally other than those used in stereo camera calibration, to test calibration accuracy. Results of both distance-based and position-based accuracy tests are returned.

Usage

```
dltTestCalibration(cal.coeff, coor.2d, nx, grid.size,
                   epipolar.reciprocal = TRUE)

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

Arguments

`cal.coeff`	a matrix of DLT calibration coefficients. The columns correspond to each camera view and the column order should match the camera view order in the fourth dimension of the `coor.2d` array.
`coor.2d`	a four-dimensional array of grid points. The first two dimensions correspond to each matrix of grid points, the third corresponds to each grid position/orientation and the fourth corresponds to each camera view. These can be read in from file by `readCheckerboardsToArray`.
`nx`	the number of points along the first dimension (e.g. this would be the number of points in each row if points in `coor.2d` are listed first by row). The number of points along the second dimension is calculated based on the total number of points per view and orientation.
`grid.size`	the size of the grid squares in real-world units (e.g. millimeters).
`epipolar.reciprocal`	a logical indicating whether epipolar distance should be calculated reciprocally and then averaged.
`object`	a list of class `"dltTestCalibration"` (the output of `dltTestCalibration()`).
`...`	further arguments passed to or from other methods.

Details

Although the RMS errors reported by `dltCalibrateCameras` can be used to assess the accuracy of a stereo camera setup, these represent how well the DLT parameters fit the calibration point set and not the reconstruction accuracy per se. It has been argued that in order to obtain a true estimation of reconstruction accuracy, an independent assessment criterion is required (Challis & Kerwin 1992). With the StereoMorph package, this is best accomplished by photographing a grid not used in the calibration and of a different square size (to test for proper scaling). These images are taken and the internal corners extracted just as in the calibration step (see `dltCalibrateCameras`), again ensuring that the test images fully sample the calibration volume and that the extracted point orders correspond between the two views. Complete instructions for each of these steps can be found at Auto-detecting checkerboard corners. The input format of `coor.2d` to `dltTestCalibration()` is the same format as the `coor.2d` input to `dltCalibrateCameras`.

`dltTestCalibration()` measures the calibration accuracy using two approaches: a distance-based approach and a position-based approach. For the distance-based approach (e.g. Tashman & Anderst 2003; Brainerd et al. 2010), random pairs of grid points are chosen (without resampling), reconstructed and the distance between the reconstructed points is compared with the actual distance. The deviations from the true distance (interpoint distance error or IPD error) for each pair of points are returned in the `ipd.error` vector. `dltTestCalibration()` also measures IPD error of only adjacent points, returned in the vector `adj.pair.ipd.error`. With a sufficient number of grid points, adjacent points are close enough that one can test how IPD error varies as a function of the distance from the approximate center of the calibrated volume (`adj.pair.centroid.dist`) or along a particular dimension (`adj.pair.mean.pos`).

One challenge in interpreting the IPD error, however, is that each deviation represents error in the x, y and z position of two points. This makes it difficult to assess the accuracy of a particular point or along a particular dimension. Since we do not know the 3D coordinates of a test grid placed at an arbitrary orientation in the calibration volume, we must find the best fit 3D position in order to assess

positional accuracy. For the position-based approach, `dltTestCalibration()` takes an ideal grid of the same square size and dimensions and optimally aligns it with the reconstructed test points using `findOptimalPointAlignment`. The reconstructed test points can then be compared with their corresponding reference points. These errors are returned in the matrix `aitr.error` (aligned ideal to reconstructed point position). This approach has the disadvantage that best fit alignment will tend to align the reference grid where the error is highest so as to minimize differences. This can decrease error where it is in actuality relatively high and vice versa.

For a step-by-step tutorial on how to use `dltTestCalibration()` see Testing the calibration accuracy.

Value

a list of class `"dltTestCalibration"` with the following elements:

`num.grids` the number of test calibration grids used in accuracy assessment.

`epipolar.error` the epipolar error (distance) for every test calibration point. This is the reciprocal epipolar distance if `epipolar.reciprocal` is `TRUE`. See `dltEpipolarDistance`.

`epipolar.rmse` the root-mean-square error of `epipolar.error`.

`ipd.error` a vector of the deviations from the true distance between random pairs of points (without resampling).

`pair.dist` a vector of the true distances between the random pairs of points in `ipd.error`.

`ipd.rmse` the root-mean-square error of `ipd.error`.

`adj.pair.ipd.error` a vector of the deviations from the true distance between random pairs of adjacent points (without resampling).

`adj.pair.mean.pos` a three-column matrix of the mean position (midpoint) of the adjacent pairs of points in `adj.pair.ipd.error`.

`adj.pair.centroid.dist` a vector of the distances from each point in `adj.pair.mean.pos` to the centroid of all `adj.pair.mean.pos`.

`aitr.error` a three-column matrix of the x, y and z position errors for the reconstructed test calibration points relative to optimally aligned ideal grid points.

`aitr.dist.error` a vector of the distances between the reconstructed test calibration points and the optimally aligned ideal grid points. Note that ideally this distance should be zero so all values in this vector are positive.

`aitr.dist.rmse` the RMS error (or deviation) of `aitr.dist.error`.

`aitr.rmse` a vector of the RMS error (or deviation) of `aitr.error` along each dimension. This is very similar to the standard deviation of `aitr.error` along each dimension.

`aitr.pos` a three-column matrix of the ideal grid points after best fit alignment to the reconstructed grid points.

`aitr.centroid.dist` a vector of the distances between each AITR point and the centroid of all AITR points.

Author(s)

Aaron Olsen

References

Challis, J.H. and Kerwin, D.G. (1992). Accuracy assessment and control point configuration when using the DLT for photogrammetry. *Journal of Biomechanics*, **25** (9), 1053–1058.

Tashman, S. and Anderst, W. (2003). *In Vivo* Measurement of Dynamic Joint Motion Using High Speed Biplane Radiography and CT: Application to Canine ACL Deficiency. *Transactions of the ASME*, **125**, 238–245.

Brainerd, E.L., Baier, D.B., Gatesy, S.M., Hedrick, T.L., Metzger, K.A., Gilbert, S.L and Crisco, J.J. (2010). X-ray reconstruction of moving morphology (XROMM): Precision, accuracy and applications in comparative biomechanics research. *Journal of Experimental Zoology*, **313A**, 262–279.

For a general overview of DLT: http://kwon3d.com/theory/dlt/dlt.html

See Also

dltCalibrateCameras, dltCoefficients, readCheckerboardsToArray, dltEpipolarDistance, findCheckerboardCorners

Examples

```
## SET NUMBER OF INTERNAL ROWS AND COLUMNS
nx <- 21
ny <- 14

## GET THE FILE DIRECTORY FOR EXTRA R PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## SET FILE PATH TO CHECKERBOARD CORNERS FROM TEST CALIBRATION IMAGE SET
file <- matrix(c(paste0(fdir, "test_cal_a", 1:11, "_v1.txt"),
  paste0(fdir, "test_cal_a", 1:11, "_v2.txt")), ncol=2)

## READ IN CHECKERBOARD CORNERS
coor.2d <- readCheckerboardsToArray(file=file, nx=nx, ny=ny, col.reverse=FALSE)

## SET GRID SIZE OF TEST CHECKERBOARDS (IN MM)
grid.size <- 4.2218

## LOAD CALIBRATION COEFFICIENTS
cal.coeff <- as.matrix(read.table(file=paste0(fdir, "cal_coeffs.txt")))

## TEST CALIBRATION ACCURACY
## USE ONLY A SUBSET (FIVE) OF TEST CALIBRATION IMAGES
## IN THE TUTORIAL POINTS, UNITS NOT IN PIXELS ARE MILLIMETERS
dlt_test <- dltTestCalibration(cal.coeff=cal.coeff, coor.2d=coor.2d[, , 1:5, ], nx=nx,
  grid.size=grid.size)

## RUN TEST ON ALL TEST CALIBRATION IMAGES
## Not run:
```

```
dlt_test <- dltTestCalibration(cal.coeff=cal.coeff, coor.2d=coor.2d, nx=nx,
  grid.size=grid.size)

## End(Not run)

## PRINT SUMMARY
summary(dlt_test)

## PLOT A HISTOGRAM OF THE INTERPOINT DISTANCE ERROR
hist(dlt_test$ipd.error)

## PLOT ADJACENT POINT DISTANCE ERROR AS A FUNCTION OF POSITION ALONG THE Y-AXIS
dev.new()
plot(dlt_test$adj.pair.ipd.error, abs(dlt_test$adj.pair.mean.pos[, 2]))

## PLOT POSITION-BASED ERROR AS A FUNCTION OF POSITION ALONG THE X-AXIS
dev.new()
plot(dlt_test$aitr.pos[, 1], abs(dlt_test$aitr.error[, 1]))
```

`dltTransformationParameterRMSError` *Returns the error during transformation parameter optimization*

Description

Returns the RMS error from `dltCoefficients` after applying a given set of transformation parameters to grid points in stereo camera calibration. This function is called internally by the function `dltCalibrateCameras` to estimate the position and orientation of a set of calibration grid points that minimizes calibration error.

Usage

```
dltTransformationParameterRMSError(p, coor.2d, nx, ny, sx, sy = NULL,
                                   p.fixed = NULL)
```

Arguments

`p`	a vector of six transformation parameters per grid. The first three being rotational parameters (rotation about the z, y and x axes, respectively) and the second three being translational parameters (translation along the x, y and z axes, respectively). For more than one grid, these six values are concatenated as a vector.
`coor.2d`	a four-dimensional array of grid points passed from `dltCalibrateCameras`.
`nx`	the number of points along the first dimension (e.g. this would be the number of points in each row if points are listed first by row).
`ny`	the number of points along the second dimension (e.g. this would be the number of points in each column if points are listed first by row).

sx	a scaling factor along the first dimension.
sy	a scaling factor along the second dimension. If the grid blocks are squares, this can be left as NULL and only sx will be used.
p.fixed	a set of transformation parameters to be appended to the beginning of p that will are fixed (constant) during the optimization step.

Value

the mean RMS error from dltCoefficients across all views.

Author(s)

Aaron Olsen

See Also

transformPlanarCalibrationCoordinates, dltCoefficients, dltCalibrateCameras

---

drawCheckerboard *Creates a checkerboard image*

Description

Creates a checkerboard image of specified dimensions and saves to an input file path. The dimensions of the checkerboard are specified by the number of internal corners (the number of squares minus one).

Usage

```
drawCheckerboard(nx, ny, square.size, filename,
                 margin.x = c(round(square.size/2), round(square.size/2)),
                 margin.y = c(round(square.size/2), round(square.size/2)),
                 ...)
```

Arguments

nx	the number of internal corners in the horizontal direction (the number of squares in each row minus one).
ny	the number of internal corners in the vertical direction (the number of squares in each column minus one).
square.size	the square size in pixels.
filename	the file path and name to which the image should be saved. The filename must be a valid image filename. Acceptable extensions are: jpg, jpeg, bmp, png and tiff.
margin.x	the margin in pixels on the left and right sides of the checkerboard pattern.
margin.y	the margin in pixels on the top and bottom of the checkerboard pattern.
...	further arguments to be passed to the image function corresponding to the extension in filename (e.g. compression, quality, etc.).

Details

This function requires the `grid` package. The image type is determined automatically from the filename and the corresponding image writing function is called.

For a step-by-step tutorial on how to use `drawCheckerboard()`, see Creating a checkerboard pattern.

Value

returns null device

Author(s)

Aaron Olsen

See Also

`readCheckerboardsToArray`

Examples

```
## NUMBER OF INTERNAL CORNERS IN THE HORIZONTAL DIMENSION
## NUMBER OF ROWS OF SQUARES MINUS ONE
nx <- 21

## NUMBER OF INTERNAL CORNERS IN THE VERTICAL DIMENSION
## NUMBER OF COLUMNS OF SQUARES MINUS ONE
ny <- 14

## SQUARE SIZE IN PIXELS
square.size <- 200

## WHERE TO SAVE THE FILE
filename <- paste0("checkerboard_", nx, "_", ny, "_", square.size, ".jpeg")

## Not run:
## DRAW CHECKERBOARD
## FILE WILL BE CREATED IN CURRENT WORKING DIRECTORY
drawCheckerboard(nx=nx, ny=ny, square.size=square.size, filename=filename)

## End(Not run)
```

---

findCheckerboardCorners

Finds internal corners of a checkerboard pattern

Description

This function finds the internal corners of a checkerboard pattern in an image.

Usage

```
findCheckerboardCorners(image.file, nx, ny, corner.file=NULL, verify.file=NULL,
                        perim.min = 50, perim.max = NULL, dilations.min = 0,
                      dilations.max = 7, sub.pix.win = NULL, poly.cont.min=-0.1,
                        poly.cont.max=0.2, quad.approx.thresh = NULL,
                        print.progress=TRUE, verbose=FALSE, debug = FALSE)
```

Arguments

image.file	File path(s) to image(s) or to folder(s) containing image(s) (and only images). The image(s) should be a JPEG and include a checkerboard pattern. Can be a vector or matrix. Many different inputs accepted, see "Examples".
nx	The number of internal corners in the checkerboard along one dimension. Note that this is not the number of squares (see "Details").
ny	The number of internal corners in the checkerboard along a second dimension.
corner.file	File path(s) to text file(s) or to folder(s) where the corners should be saved. Can be a vector or matrix. If NULL, corners are not saved to a text file. Many different inputs accepted, see "Examples".
verify.file	File path(s) to JPEG image(s) or to folder(s) where verification images should be saved. Can be a vector or matrix. If NULL, verification images are not created. Many different inputs accepted, see "Examples".
perim.min	The minimum expected perimeter of a black square in the checkerboard pattern (in pixels).
perim.max	The maximum expected perimeter of a black square in the checkerboard pattern (in pixels).
dilations.min	The initial number of dilations to perform on the image. See "Details".
dilations.max	The maximum number of dilations to perform on the image. If equal to dilations.min, the function will only perform one dilation. See "details".
sub.pix.win	The window size to use in determining the corner positions to subpixel resolution. If NULL, this is determined automatically based on the size of the found corners.
poly.cont.min	The minimum allowed aspect ratio of the polygon contours, used as a threshold in identifying quadrangles.
poly.cont.max	The maximum allowed aspect ratio of the polygon contours, used as a threshold in identifying quadrangles.
quad.approx.thresh	A threshold for the perimeter of black squares in which method to use to approximate the shape as a quadrangle.
print.progress	Logical indicating whether the function progress should be printed to the console. See verbose.
verbose	Logical indicating whether more detailed progress reports to the console. If verbose is FALSE, only the image name and whether the corners were found successfully are printed. If verbose is TRUE, the outcome of the corner search at the conclusion of each dilation is also printed.

debug — Logical indicating whether images should be created at each of several steps in the corner search. These will be written to the same location as the images written to `verify.file`. If `debug` is `TRUE`, `verify.file` must be defined. Additionally, `dilations.min` and `dilations.max` should be identical since debugging images are created at each dilation and will be overwritten if a range of dilations is input.

Details

This function automatically detects checkerboard corners in an image and returns the pixel coordinates of the internal corners (where the corners of the black squares contact other black squares) to subpixel resolution. The function uses several C++ functions for image processing written by the author and compiled with the StereoMorph package but hidden until documentation can be written for more general use. Currently the function only works with JPEG images (.jpg or .jpeg); this is the most common digital camera image format output. For large images (10-20 MB), the function can take from 5-15 seconds per image.

`image.file` input to the function can be of several different forms. First, it can be file paths to particular images or file paths to a folder or folders containing images. Secondly, it can be in a vector or matrix format. The format of `image.file` will dictate the structure of the value returned by the function. If a single image file is input, a two-colum matrix of corners (where the two columns correspond to the x, y pixel coordinates) is returned. If the input is a vector of file paths or folders containing images, a three-dimensional array is returned; the first two dimensions are the rows and columns of each corner matrix and the third dimension is the order of the corresponding image files in `image.file`. If the input is a matrix of file paths or folders containing images, a four-dimensional array is returned; the first two dimensions are the rows and columns of each corner matrix and the third and fourth dimensions are the positions of the corresponding image files in the `image.file` matrix. If `image.file` is a folder or folders containing images, the folders cannot contain any other files.

The inputs `corner.file` and `verify.file` are optional but if they are non-`NULL`, they should be of the same format as `image.file`. If `image.file` is a folder or folders containing images, folders can also be input for `corner.file` and `verify.file`. In this case, the function will automatically name the corner files and verify image files with the same names as the images and as text files and JPEG files, respectively. The corners are saved to a text file as a two column matrix without a header or row names.

For every input image, the function begins by reading in the image (using `readJPEG()` of the 'jpeg' package). For large images this is one of the most time-consuming steps. The image is converted to grayscale using the internal function `rgbToGray()`. The image is thresholded to create a binary image (black and white) based on an adaptive threshold. The threshold is created using the internal function `meanBlurImage()` and the image thresholded with the internal function `thresholdImageMatrix()`. Morphological closing is performed to reduce noise using the internal functions `dilateImage()` and `erodeImage()`.

The function then proceeds to dilate the image (expand white areas and consolidate black areas) using a 3x3 square kernel for the range specified by `dilations.min` and `dilations.max`. This separates the black squares from each other so that their perimeters can be detected as separate contours. For each dilation, all edge points are identified (black pixels with a neighboring white pixel and vice versa) using the internal function `findBoundaryPoints()`. Contours (connected edge points) are identified by the internal function `generateQuads()`, retaining only contours that

are quadrangles. The midpoints between adjoining corners of all the quads are found using the internal function `intCornersFromQuads()`; among these will be the full set of internal corners.

If the initial set of internal corners exceeds the expectation, the internal corners are filtered, fitting a line to the internal corner set and removing the points at the furthest difference from the line of best fit until the number of corners matches the expectation. The filtered internal corner set is then ordered using the internal function `orderCorners()` so that first corner is the top left most corner in the pattern and the sequence of internal corners proceeds along `nx` first and `ny` second. Lastly, the function finds the internal corner positions to subpixel resolution (using the internal function `findCornerSubPix()`) by sampling a window around the approximate location of the internal corners (of dimensions determined by `sub.pix.win`) to find a point optimally positioned at the intersection of diagonally opposing white and black squares. If determined automatically, this sampling window will usually be 23x23 pixels. It is the sampling of this large image region that allows the function to return the corner position to subpixel resolution.

If `verify.file` is non-`NULL`, the internal corners are overlayed on the input image to verify that the correct corners have been found and in the correct order. The first corner is circled in red, a green line interconnects all the intermediate corners in sequence and the last corner is circled in blue (the order of colors then being RGB).

Value

An array of the pixel coordinates of internal corners to subpixel resolution in an array of two (one checkerboard input), three (if `image.file` is a vector) or four dimensions (if `image.file` is a matrix). For images in which the expected number of internal corners were not found, an `NA` matrix is returned for those particular images. The corners are returned along the `nx` dimension first and the `ny` dimension second.

Author(s)

Aaron Olsen

References

This function was written based on the methodology described in 'Learning OpenCV' for the automated detection of internal checkerboard corners (Bradski and Kaehler 2008).

See Also

`readCheckerboardsToArray`, `measureCheckerboardSize`

Examples

```
## GET THE FILE DIRECTORY FOR EXTRA R PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## FIND 5 X 3 INTERNAL CORNERS IN A SINGLE IMAGE
corners <- findCheckerboardCorners(image.file=paste0(fdir,
     "Checkerboards/RUlna.JPG"), perim.min=180, nx=5, ny=3)

## FIND 5 X 3 INTERNAL CORNERS IN ALL IMAGES IN A FOLDER (HERE 3)
```

```
corners <- findCheckerboardCorners(image.file=paste0(fdir,
     "Checkerboards"), perim.min=180, nx=5, ny=3)

## WHICH DIMENSIONS ARE ASSIGNED TO NX AND NY IS ARBITRARY BUT REVERSING
## THESE WILL CHANGE THE SEQUENCE IN WHICH THE CORNERS ARE RETURNED
corners <- findCheckerboardCorners(image.file=paste0(fdir,
     "Checkerboards/RUlna.JPG"), perim.min=180, nx=3, ny=5)
```

`findOptimalPointAlignment` *Optimally aligns one point set to another*

Description

This function translates and rotates one point set, optimally aligning it with another point set.

Usage

```
findOptimalPointAlignment(m1, m2)
```

Arguments

`m1`	a point set matrix
`m2`	a second point set matrix of the same dimensions as `m1`

Details

This function optimally aligns point set `m2` with point set `m1`. `m1` and `m2` must contain the exact same landmarks or points in the same order. Points present in `m2` but not `m1` should be `NA` in `m1`. They do not need to be `NA` in `m2`; all translations and rotations will be applied to all points in `m2` even though only shared points will be used in the alignment.

The function first centers the centroid `m2` about the centroid of `m1`. The function `svd()` is then used to find the 3D rotation matrix that optimally aligns `m2` to `m1` based on common points. The positions of points in `m2` relative to one another are unchanged. Thus, optimal rotation is constrained to already translated point sets. Depending on the point sets, a better alignment may be possible by allowing translation and rotation to be optimized simultaneously.

This function is called by `unifyLandmarks` to align landmark sets and by `dltTestCalibration` to test accuracy in reconstructed calibration grids.

Value

`m2` after alignment.

Note

Modified from `unifyVD()` by Annat Haber.

Author(s)

Annat Haber, Aaron Olsen

References

Rohlf, F.J. (1990) "Chapter 10. Rotational fit (Procrustes) Methods." *Proceedings of the Michigan Morphometrics Workshop*. Ed. F. James Rohlf and Fred L. Bookstein. The University of Michigan Museum of Zoology, 1990. 227–236. Info page at lib.umich.edu

See Also

unifyLandmarks

Examples

```
## MAKE MATRIX OF 3D POINTS
m1 <- matrix(c(0,0,0, 1,3,2, 4,2,1, 5,5,3, 1,4,2, 3,6,4), nrow=6, ncol=3)

## COPY TO M2
m2 <- m1

## MAKE MISSING POINT IN M1
## ALTHOUGH NOT USED IN THE ALIGNMENT THE CORRESPONDING POINT
##  IN M2 IS STILL RETURNED AFTER ALIGNMENT
m1[3, ] <- NA

## CENTER M2 ABOUT CE
m2 <- m2 %*% rotationMatrixZYX(pi/6, -pi/3, pi/8)

## TRANSLATE M2
m2 <- m2 + matrix(c(2,3,4), nrow=6, ncol=3, byrow=TRUE)

## ALIGN M2 TO M1
m3 <- findOptimalPointAlignment(m1, m2)

## NOTE THAT RETURNED MATRIX IS IDENTICAL TO M1
## OF COURSE REAL WORLD DATA WILL HAVE SOME ERROR
m1
m3
```

---

gridPointsFit *Fits regularly spaced points to a sample line or grid*

Description

This function is used to fit a model of points at a regular interval to a sample of points in one dimension. The function is used by measureCheckerboardSize to estimate the solution to the inter-point distance of points along a line or in a grid.

Usage

```
gridPointsFit(p, nx, ny=NULL)
```

Arguments

p	The parameters defining the regular point distribution. When nx is NULL, p is of length 2. When nx is non-NULL, p is of length 3.
nx	The number of points to be created at regular spacing along one dimension.
ny	The number of points to be created at regular spacing along a second dimension.

Details

This function is used to fit a model of points at a regular interval to a sample of points in one dimension. The function is used by measureCheckerboardSize to estimate the solution to the inter-point distance of points along a line or in a grid. To fit a model to points along lines and grids in two dimensions, each dimension is fit separately. A best fit estimate of the true interval between points can then be calculated from the optimized parameters. See the examples below for how to use gridPointsFit() to estimate the inter-point intervals of line and grid points.

Value

a vector of length nx*ny.

Author(s)

Aaron Olsen

See Also

measureCheckerboardSize

Examples

```
## ESTIMATE LINE INTER-POINT INTERVAL
# GENERATE POINTS AT A REGULAR INTERVAL WITH NORMAL, RANDOM VARIATION
pts <- cbind((1:500) + rnorm(500, sd=1), (1:500) + rnorm(500, sd=1))

# FIND THE MEAN SUCCESSIVE POINT-TO-DISTANCE
# NOTE THAT THIS CONSISTENTLY OVERESTIMATES THE TRUE INTERVAL
mean(sqrt(rowSums((pts[2:nrow(pts), ] - pts[1:(nrow(pts)-1), ])^2)))

# FIT A REGULARLY SPACED POINTS MODEL TO EACH DIMENSION OF THE POINTS MATRIX
fit_x <- nlminb(start=c(pts[1, 1], pts[2, 1]-pts[1, 1]),
    objective=gridPointsFitError, nx=nrow(pts), points=pts[, 1])
fit_y <- nlminb(start=c(pts[1, 2], pts[2, 2]-pts[1, 2]),
    objective=gridPointsFitError, nx=nrow(pts), points=pts[, 2])

# FIND THE BEST FIT INTER-POINT DISTANCE
```

```
# MORE ACCURATELY RECOVERS TRUE INTERVAL
sqrt(fit_x$par[2]^2 + fit_y$par[2]^2)


## ESTIMATE REGULAR GRID SQUARE SIZE
# GENERATE A REGULAR GRID WITH NORMAL, RANDOM VARIATION
corners <- cbind(
    rep(1:20, 20) + rnorm(20^2, sd=0.1),
    c(t(matrix(1:20, nrow=20, ncol=20))) + rnorm(20^2, sd=0.1))

# FIT A REGULARLY SPACED POINTS MODEL TO EACH DIMENSION OF THE POINTS MATRIX
fit_x <- nlminb(
    start=c(corners[1, 1], corners[2, 1]-corners[1, 1], 0),
    objective=gridPointsFitError, points=corners[, 1], nx=20, ny=20)
fit_y <- nlminb(
    start=c(corners[1, 2], corners[2, 2]-corners[1, 2], 0),
    objective=gridPointsFitError, points=corners[, 2], nx=20, ny=20)

# FIND THE BEST FIT INTER-POINT DISTANCE (SQUARE SIZE)
sqrt(fit_x$par[2]^2 + fit_y$par[2]^2)
```

`imagePlaneGridTransform`

Performs image perspective transformations to a grid

Description

This function takes parameters describing a 3D planar grid projected onto a 2D image plane and returns a grid of specified dimensions. Users will probably not call this function directly. Rather, it is used by `resampleGridImagePoints` to produce grid points with the same transformations as an imaged grid but with fewer points.

Usage

```
imagePlaneGridTransform(p, nx, ny)
```

Arguments

`p`	a vector of 12 grid parameters. The first eight values are the x,y-coordinates of the four grid corners (`x1`, `y1`, `x2`, `y2`, etc.) and the last four values are transformation parameters for slope and interpoint spacing.
`nx`	the number of points along the first dimension. Note that although the grid can have a different number of rows than columns, the grid units themselves should be square (of uniform size in both dimensions).
`ny`	the number of points along the second dimension.

Details

When taking a photo of planar grid points (such as the internal corners of a checkerboard pattern) arbitrarily oriented in 3D space, the distribution of grid points in a 2D photograph will reflect several transformations. The grid may be translated to any position within the image plane and rotated by any angle. Additionally, if the grid plane is not parallel to the image plane, perspective effects will cause points further away to appear closer together. When arbitrary 3D position and perspective effects are combined, the transformation of a planar grid can be quite extreme (see example).

`imagePlaneGridTransform()` takes 12 parameters describing these effects and applies them to a grid of the specified dimensions, returning the transformed grid points. The first eight parameters are the x,y-coordinates of the four grid corners (`x1`, `y1`, `x2`, `y2`, etc.). The ninth and tenth parameters describe how interpoint spacing changes from row-to-row and column-to-column, respectively. This is the `a` parameter in the function `quadraticPointsOnInterval`. A value of zero indicates uniform spacing between consecutive points across all rows while values less or greater than zero indicates points that become closer together or further apart from one row or column to the next. The eleventh and twelfth parameters are analogous to the ninth and tenth parameters but describe how spacing changes between rows and columns instead of between points. These last two parameters are also the `a` parameter in the function `quadraticPointsOnInterval`.

Currently, `imagePlaneGridTransform()` does not currently account for lens distortion (e.g. barrel, pincushion, etc.). If distortion is significant, users should undistort the photographs prior to using `imagePlaneGridTransform()`. It is hoped that future versions will include additional parameters to account for lens distortion.

Users will probably not call `imagePlaneGridTransform()` directly. In this package, this function is used by `resampleGridImagePoints` to both fit transformation parameters to a matrix of imaged grid points and to produce a transformed grid consisting of fewer points. In this way, fewer points (but representing the same amount of information) can be used in more computationally intensive steps.

Value

a matrix of transformed grid points.

Author(s)

Aaron Olsen

See Also

`resampleGridImagePoints`, `quadraticPointsOnInterval`, `imagePlaneGridTransformError`

Examples

```
## SET GRID PARAMETERS
## THE FIRST 8 NUMBERS ARE CORNERS
## THE LAST 4 NUMBERS ARE TRANSFORMATION PARAMETERS
p <- c(3656, 379, 707, 264, 383, 1034, 3984, 1164, 63.772, -25.211, -0.818, -3.339)

## CREATE TRANSFORMED GRID
grid <- imagePlaneGridTransform(p=p, nx=21, ny=14)
```

```
## PLOT GRID
plot(grid)

## MARK CORNERS OF GRID FROM p
points(matrix(p, nrow=4, ncol=2, byrow=TRUE), col='red', lwd=2, cex=1.5)
```

`imagePlaneGridTransformError` *Returns imagePlaneGridTransform error*

Description

Returns the mean error between a matrix of grid points and a matrix of transformed grid points (produced by `imagePlaneGridTransform`). This function is called internally by the function `resampleGridImagePoints` in evaluating the goodness of fit between imaged grid points and grid points produced by an image perspective model.

Usage

```
imagePlaneGridTransformError(p, nx, ny, grid)
```

Arguments

`p`	a vector of 12 grid parameters (see `imagePlaneGridTransform`).
`nx`	the number of points along the first dimension.
`ny`	the number of points along the second dimension.
`grid`	a matrix of grid points to be compared against the model grid points.

Value

the mean error.

Author(s)

Aaron Olsen

See Also

`imagePlaneGridTransform`, `resampleGridImagePoints`

`landmarkListToMatrix` *Converts a landmark list to a landmark matrix*

Description

Converts a landmark list to a landmark matrix. The landmark matrix is identical to the matrix that would be returned if the landmark files were sent directly to `readLandmarksToMatrix`.

Usage

```
landmarkListToMatrix(lm.list)
```

Arguments

`lm.list`	a landmark list. See `readLandmarksToList`.

Value

a landmark matrix.

Author(s)

Aaron Olsen

See Also

`readLandmarksToList`, `readLandmarksToMatrix`

Examples

```
## GET FILE DIRECTORY FOR PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## SET FILES TO LOAD - TWO DIFFERENT 3D POINT SETS
file <- paste0(fdir, "lm_3d_even_a", 1:2, ".txt")

## READ LANDMARKS INTO A LIST
lm.list <- readLandmarksToList(file=file, row.names=1)

## CONVERT LANDMARK LIST TO LANDMARK MATRIX
lm.matrix <- landmarkListToMatrix(lm.list)

lm.matrix
```

`landmarkMatrixToList` *Converts a landmark matrix to a landmark list*

Description

Converts a landmark matrix to a landmark list.

Usage

```
landmarkMatrixToList(lm.matrix, semilandmark.pattern='[0-9]+$', k=ncol(lm.matrix))
```

Arguments

`lm.matrix`	a landmark matrix. See `readLandmarksToMatrix`.
`semilandmark.pattern`	a regular expression pattern passed to `sub()` for identifying and grouping curve points. The default is landmark names ending in one or more numbers. To disable grouping, set to code".
`k`	the number of dimensions of the landmark data.

Value

a landmark list.

Author(s)

Aaron Olsen

See Also

`landmarkListToMatrix`, `readLandmarksToList`, `readLandmarksToMatrix`

Examples

```
## GET FILE DIRECTORY FOR PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## SET FILES TO LOAD - TWO DIFFERENT 3D POINT SETS
file <- paste0(fdir, "lm_2d_a1_v", 1:2, ".txt")

## READ LANDMARKS INTO A LIST
lm.matrix <- readLandmarksToMatrix(file=file, row.names=1)

## CONVERT LANDMARK LIST TO LANDMARK MATRIX
lm.list <- landmarkMatrixToList(lm.matrix, k=2)

## CAN BE CONVERTED BACK INTO MATRIX
## RECOVERING THE SAME MATRIX AS THE ORIGINAL
lm.matrix <- landmarkListToMatrix(lm.list)
```

`measureCheckerboardSize` *Estimates checkerboard square size*

Description

This function estimates the square size of a checkerboard, optionally scaling this to real-world units (e.g. millimeters).

Usage

```
measureCheckerboardSize(corner.file, nx, ruler.file=NULL, ruler.pt.size=NULL)

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

Arguments

`corner.file`	a file path to text file containing a matrix of internal corners from a checkerboard pattern (a point grid) or the matrix itself. The text file must not have row names or a header.
`nx`	the number of internal corners in the first dimension along which the checkerboard points are ordered.
`ruler.file`	a file path to a text file containing a matrix of evenly spaced points digitized along a ruler (or comparable standard) or the matrix itself. The text file must have row names but no header or column names.
`ruler.pt.size`	the size of the spacing between points in the `ruler.file` matrix in real world units. This can be numeric or alphanumeric including the unit (see "Details").
`object`	a list of class `"measureCheckerboardSize"`.
`...`	further arguments passed to other methods.

Details

`corner.file` can be a file path to a text file containing a matrix of internal corners from a checkerboard pattern (ie points in a regular grid pattern) or the matrix itself. These can be automatically detected from a JPEG image using the function `findCheckerboardCorners`. The function first fits a camera perspective model to the corner points to robustly compare the opposing side lengths of the grid (see `resampleGridImagePoints`). These are returned as `side.lengths` and are displayed when calling the summary method. Opposing sides that differ greatly in length indicate that the grid was not completely flat relative to the image plane when it was photographed.

`measureCheckerboardSize()` then estimates the checkerboard or grid square size by fitting a simple grid model to the points (see `gridPointsFit`). The best fitting parameters are used to estimate the square size. Model fitting is more robust to noise in the grid point coordinates than taking the mean inter-point distance, for instance. The model goodness of fit can be assessed by the returned elements `dist.corner.fit.mean` and `dist.corner.fit.sd`.

ruler.file can be a file path to a text file containing a matrix of points at equal intervals along a ruler or the matrix itself. These ruler points can be digitized from an image using the function digitizeImage. If ruler.file is NULL, then only the checkerboard square size (in the input units) is returned. All other return values are NULL. If ruler.file is non-NULL, the distance between consecutive ruler points (the ruler point interval) is estimated by fitting a model of points at a regular interval along a line (see gridPointsFit). The goodness of fit for the ruler point model can be assessed by the returned elements dist.ruler.fit.mean and dist.ruler.fit.sd. The estimated ruler point interval is used to scale the checkerboard square size to the units of ruler.pt.size.

ruler.pt.size can be numeric or alphanumeric (including the units). For example, '1', '1 mm' and '1.0 mm' are all possible inputs to ruler.pt.size. The units are automatically extracted and only used in the summary function to help interpret the function results. measureCheckerboardSize() also returns the estimated real-world size of a pixel. This represents the resolution of the camera at the surface of the checkerboard pattern.

See Measuring checkerboard square size for a tutorial on this function.

See Auto-detecting checkerboard corners for a tutorial on how to automatically detect checkerboard corners from an image.

Value

a list of class "measureCheckerboardSize" with the following elements:

side.lengths the lengths of the four sides of the grid estimated by camera perspective model fitting.

dist.corner.fit.mean
: the mean difference between the corner points corner.file and those generated assuming the best-fit simple grid model.

dist.corner.fit.sd
: the standard deviation in the difference between the corner points corner.file and those generated assuming the best-fit model.

square.size.px the best-fit estimate of the checkerboard square size in pixels.

square.size.rwu
: the best-fit estimate of the checkerboard square size in real-world units. NULL if ruler.file is NULL.

dist.ruler.fit.mean
: the mean difference between the ruler.file matrix and those generated assuming the best-fit model. NULL if ruler.file is NULL.

dist.ruler.fit.sd
: the standard deviation in the difference between the ruler.file matrix and those generated assuming the best-fit model. NULL if ruler.file is NULL.

ruler.size.px the best-fit estimate of the distance between consecutive points on the ruler (in pixels) in the plane of the imaged grid. NULL if ruler.file is NULL.

rwu.per.px the real-world size of a pixel in the image (the length of one side of the pixel) in the plane of the imaged grid. NULL if ruler.file is NULL.

unit if ruler.pt.size includes a unit, the unit. NULL if ruler.file is NULL.

Author(s)

Aaron Olsen

See Also

drawCheckerboard, resampleGridImagePoints, gridPointsFit, digitizeImage

Examples

```
## GET THE FILE DIRECTORY FOR EXTRA R PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## SET FILE PATH TO CHECKERBOARD POINTS FILE
corner_file <- paste0(fdir, "checker_21_14_200(9).txt")

## NUMBER OF INTERNAL CORNERS IN THE HORIZONTAL DIMENSION
nx <- 21

## NUMBER OF INTERNAL CORNERS IN THE VERTICAL DIMENSION
ny <- 14

## SET FILE PATH TO RULER POINTS FILE
ruler_file <- paste0(fdir, "ruler_21_14_200(9).txt")

## ESTIMATE SQUARE SIZE
square_size <- measureCheckerboardSize(corner.file=corner_file, nx=nx)

## PRINT SUMMARY
summary(square_size)



## ESTIMATE SQUARE SIZE AND SCALE WITH RULER POINTS
square_size_scale <- measureCheckerboardSize(corner.file=corner_file, nx=nx,
    ruler.file=ruler_file, ruler.pt.size='1 mm')

## PRINT SUMMARY
summary(square_size_scale)



## INPUT MATRICES DIRECTLY
## READ POINTS INTO MATRICES
corner_pts <- as.matrix(read.table(corner_file))
ruler_pts <- as.matrix(read.table(ruler_file, row.names=1))

## ESTIMATE SQUARE SIZE AND SCALE WITH RULER POINTS
square_size_scale <- measureCheckerboardSize(corner.file=corner_pts, nx=nx,
    ruler.file=ruler_pts, ruler.pt.size='1 mm')
```

orthogonalProjectionToLine *Finds the orthogonal projection of a point onto a line*

Description

Given a 2D or 3D input point p and a 2D or 3D line, this function finds a point on the line at a minimum distance from point p. This is equivalent to the orthogonal projection of point p onto the line.

Usage

```
orthogonalProjectionToLine(p, l1 = NULL, l2 = NULL)
```

Arguments

p	a vector of a single point or a matrix of multiple points
l1	a vector describing a point on a line or a list with line constants
l2	if l1 is a point, a second point on a line

Details

If p is a vector, the function returns a point as a vector of the same dimension. If p is a matrix, each row is treated as a point and the orthogonal projection is returned for each. These points are returned as a matrix (of the same dimension), each row being the orthogonal projection of the corresponding row in p.

The line input can be defined using one of three standard ways: two points on the line, 'm' and 'b' constants (slope and y-intercept) and direction numbers 'abc' (a vector parallel to a line through the origin). If l1 is a vector, this is taken as one point on the line and l2 must be a second point on the line. If l1 is a list, the named objects must correspond to one of these three line definitions. Two points on the line are defined as l1$l1 and l1$l2. 'm' and 'b' are defined as l1$m and l1$b. And the direction numbers 'abc' are defined as l1$a, l1$b and l1$c.

Value

a vector if p is a vector and a matrix if p is a matrix. The returned vector or matrix will be of the same dimensions as p.

Author(s)

Aaron Olsen

References

http://paulbourke.net/geometry/pointlineplane/

See Also

distancePointToLine

Examples

```
## POINT INPUT: 2D VECTOR
## LINE INPUT: l1, l2
## LINE THROUGH THE ORIGIN WITH SLOPE OF ONE
orthogonalProjectionToLine(p=c(0, 5), l1=c(0, 0), l2=c(3, 3))

## POINT INPUT: 2D VECTOR
## LINE INPUT: LIST WITH l1, l2
orthogonalProjectionToLine(p=c(0, 5), l1=list(l1=c(0, 0), l2=c(3, 3)))

## POINT INPUT: 2D VECTOR
## LINE INPUT: LIST WITH m, b
## LINE WITH Y-INTERCEPT AT ONE AND SLOPE OF ONE
orthogonalProjectionToLine(p=c(0, 5), l1=list(m=1, b=0))

## POINT INPUT: 2D VECTOR
## LINE INPUT: LIST WITH VECTOR PARALLEL TO LINE THROUGH THE ORIGIN
## LINE THROUGH THE ORIGIN WITH SLOPE OF ONE
orthogonalProjectionToLine(p=c(0, 5), l1=list(a=1, b=-1, c=0))

## POINT INPUT: 2D VECTOR
## LINE INPUT: SAME AS PREVIOUS BUT WITH Z-AXIS COMPONENT
orthogonalProjectionToLine(p=c(0, 5), l1=list(a=1, b=-1, c=1))

## POINT INPUT: 3D VECTOR
## LINE INPUT: l1, l2
orthogonalProjectionToLine(p=c(0, 5, 0), l1=list(l1=c(0, 0, 0), l2=c(3, 3, 3)))

## POINT INPUT: 2D MATRIX
## LINE INPUT: l1, l2
p <- matrix(c(0,5, 0,10), nrow=2, byrow=TRUE)
orthogonalProjectionToLine(p=p, l1=list(l1=c(0, 0), l2=c(3, 3)))

## POINT INPUT: 3D MATRIX
## LINE INPUT: l1, l2
p <- matrix(c(0,5,0, 0,10,0), nrow=2, byrow=TRUE)
orthogonalProjectionToLine(p=p, l1=list(l1=c(0, 0, 0), l2=c(3, 3, 3)))
```

---

pointsAtEvenSpacing *Generates evenly spaced points from point matrix*

Description

This function takes a matrix of points, calculates the cumulative distance from start to end and then uses the cumulative distance and intermediate points to generate evenly spaced points between the

start and end points. Linear interpolation is used between neighboring points, so the returned points will either coincide with the input points or fall on straight lines between consecutive points.

Usage

```
pointsAtEvenSpacing(x, n)
```

Arguments

x	a matrix or landmark list of points of any number of dimensions. If input is a list, only the first element is used.
n	the number of points to generate, including the start and end points.

Details

The function first removes all NA values. Then, the cumulative distance is calculated from the first to last point. The last value is taken as the total length of the line or curve, defined by matrix x. This total length is divided by n-1 to find a uniform segment length that will separate n evenly spaced points, including the first and last non-NA values in x.

The function iterates through x, finding the point that is at a distance equal to or just less than the segment length from the previous point. If the selected point is at a distance less than the segment length from the previous point, a point is chosen on the line between this point and the next to complete the full segment length. In this way, returned points will either coincide with the input points or fall on straight lines between consecutive points.

In the simplest implementation, pointsAtEvenSpacing() can be used for linear interpolation (see first example below). Define the start and end points in x as a two-row matrix and then select the number of points to include on the line.

If x represents densely sampled points on a curve (see the second example below) and if the curve can be approximated by straight lines between consecutive points, then pointsAtEvenSpacing() will provide comparable results to other methods, such as function fitting. This is especially useful for curves not easily fit by a mathematical function.

For examples on how to use pointsAtEvenSpacing() see Reconstructing 2D points and curves into 3D.

Value

a matrix of n points. The start and end points correspond to the first and last non-NA values in x.

Author(s)

Aaron Olsen

See Also

imagePlaneGridTransform, resampleGridImagePoints, imagePlaneGridTransformError

Examples

```
## LINEAR INTERPOLATION ##
## CREATE A MATRIX OF TWO POINTS
two_points <- matrix(c(0, 10, 0, 10), nrow=2, ncol=2)

## GENERATE 20 POINTS ALONG THE LINE
pts_aes <- pointsAtEvenSpacing(x=two_points, n=20)

## PLOT THE LINE
plot(two_points, type='l')

## AND THE POINTS ALONG THE LINE
points(pts_aes, col='red')


## POINTS ALONG A CURVE ##
## GET FILE DIRECTORY FOR PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## GET 3D LANDMARK AND CURVE POINT FILE AND READ INTO A MATRIX
lm.matrix <- readLandmarksToMatrix(paste0(fdir, "lm_3d_a2.txt"), row.names=1)

## PLOT THE LANDMARKS AND CURVE POINTS
pts <- na.omit(lm.matrix)
r <- abs(apply(pts, 2, 'max') - apply(pts, 2, 'min'))

## Not run:
## PLOT USING THE RGL PACKAGE
plot3d(pts, aspect=c(r[1]/r[3], r[2]/r[3], 1), size=0.5)

## End(Not run)

## CONVERT LANDMARKS TO LIST FORMAT TO EASILY ACCESS CURVE POINTS
lm.list <- landmarkMatrixToList(lm.matrix)

## CREATE 10 EVENLY SPACED POINTS ALONG ONE CURVE
lm.list$pterygoid_crest_R <- pointsAtEvenSpacing(x=lm.list$pterygoid_crest_R, n=10)

## CREATE 15 ALONG ANOTHER
lm.list$tomium_R <- pointsAtEvenSpacing(x=lm.list$tomium_R, n=15)

## CONVERT BACK TO MATRIX
lm.matrix <- landmarkListToMatrix(lm.list)

## Not run:
## PLOT NEW EVENLY SPACED POINTS WITH PREVIOUS POINTS
plot3d(lm.matrix, add=T, size=4, col='red')

## End(Not run)
```

quadraticPointsOnInterval
Generates points along an interval with quadratic parameterization

Description

Generates a specified number of points on an interval, applying a quadratic function to interpoint spacing. This function is called internally by imagePlaneGridTransform.

Usage

```
quadraticPointsOnInterval(t1, t2, n, a)
```

Arguments

t1	the starting value of the returned points.
t2	the final value of the returned points.
n	the number of points.
a	a quadratic parameter describing how interpoint spacing changes over the interval.

Details

The parameter a describes how strong of a skew to place on the interpoint distances over the interval specified by t1 and t2. When a=0, the points are spaced uniformly across the interval. When a>0 or a<0, points become further apart or closer together along the interval, respectively, at the rate of a quadratic function (see "Examples").

Value

a vector of points.

Author(s)

Aaron Olsen

See Also

imagePlaneGridTransform, resampleGridImagePoints, imagePlaneGridTransformError

Examples

```
## GENERATE EVENLY SPACED POINTS ON INTERVAL
q0 <- quadraticPointsOnInterval(t1=0, t2=1, n=10, a=0)

## MAKE POINTS PROGRESSIVELY FURTHER APART ALONG INTERVAL
qgt0 <- quadraticPointsOnInterval(t1=0, t2=1, n=10, a=1)
```

```
## MAKE POINTS PROGRESSIVELY CLOSER TOGETHER ALONG INTERVAL
qlt0 <- quadraticPointsOnInterval(t1=0, t2=1, n=10, a=-1)

## PLOT POINTS ON THREE SEPARATE LINES
plot(q0, rep(0, 10))
points(qgt0, rep(0.5, 10), col='green')
points(qlt0, rep(-0.5, 10), col='blue')
```

`readBezierControlPoints` *Reads a file of Bezier control points*

Description

Reads Bezier control points from a file or files into a list grouped first by curve name and then by the index of the file from which they were read. A separate function from the standard read functions is necessary since the number of control points may differ for each Bezier curve or spline and, thus, the number of values may differ by row.

Usage

```
readBezierControlPoints(file, ndim = 2, ...)
```

Arguments

`file`	file(s) to be read.
`ndim`	the number of dimensions of the Bezier curve points
`...`	further arguments to be passed to `readLines()`.

Details

The rows of each file must start with the name of the curve or spline followed by the control points, all separated by tabs. The control points are listed first by dimension and then by point (`x1\ty1\tx2\ty2` etc.). For example, three Bezier points starting with [100, 200] would be on one line as follows, with `\t` replaced by tabs.

`tomium_R\t100\t200\t300\t100\t400\t300`

Each Bezier curve or spline is first grouped into a list by curve name (e.g. `list$tomium_R`) and then by the index of the file from which it was read (e.g. `list$tomium_R[[1]]` from the first file). The control points are made into a matrix where the number of columns corresponds to `ndim`. The Bezier list structure is similar to the landmark list structure created by `readLandmarksToList` and can be used to generate points along a Bezier curve or spline. See the R package bezier for more details.

Value

a list of Bezier control points grouped by name and file number.

Author(s)

Aaron Olsen

See Also

readLandmarksToArray, readCheckerboardsToArray, readLandmarksToList, readLandmarksToMatrix

Examples

```
## GET FILE DIRECTORY FOR PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## FILE TO READ
file <- paste0(fdir, "bezier_control_points_a2_v", 1:2, ".txt")

## FILE TO READ
bcp <- readBezierControlPoints(file=file)
```

---

readCheckerboardsToArray

Reads file(s) containing grid points into an array

Description

This function reads grid point matrices into an array from a matrix files allowing for point order reversals along rows, columns, or both.

Usage

```
readCheckerboardsToArray(file, nx, ny, col.reverse = FALSE, row.reverse = FALSE,
                         na.omit=FALSE, ...)
```

Arguments

file	a matrix of file paths to be read into an array. Each file path should correspond to a file containing a single landmark matrix.
nx	the number of internal corners in the first dimension along which grid points are ordered.
ny	the number of internal corners in the second dimension along which grid points are ordered.
col.reverse	a logical indicating whether the column order of grid points should be reversed. Can be either single value, a vector or a matrix.
row.reverse	a logical indicating whether the row order of grid points should be reversed. Can be either single value, a vector or a matrix.
na.omit	whether landmarks with NA values in any file should be omitted.
...	further arguments to be passed to readLandmarksToArray().

Details

When using planar grid points to find an optimal stereo calibration, ensuring that the grid point coordinates are listed in the same order from different camera views is challenging. When cameras are viewing the same points from different orientations (e.g. one camera is upside-down relative to another) and when the checkerboard itself is in different orientations, columns and/or rows in one grid point matrix could be flipped relative to another camera view.

`readCheckerboardsToArray()` enables correction for this by allowing users to specify whether the rows, columns or both should be reversed after the points are read from a file. If the checkerboard changes orientation within a single camera view it could be necessary to specify row and/or column reversals individually for each file. `col.reverse` and `row.reverse` can both be either a single logical, a vector of logicals or a matrix of logicals. This allows `col.reverse` and `row.reverse` to be specified for all files in a vector or matrix or for each file separately. Vector inputs of `col.reverse` and `row.reverse` with a matrix input of `file` will be applied to each column of `file` (see last example below).

Row reversal means that point order is reversed along the second dimension (the order along the first dimension is kept intact). Column reversal means that point order is reversed along the first dimension (the order along the second dimension is kept intact). Setting both `col.reverse` and `row.reverse` to `TRUE` is equivalent to reversing the order of points from start to end (row and column structures have no effect). These operations are perhaps best understood through the examples below. For an example of the grid point ordering scheme, also see `distanceGridUnits`.

For a step-by-step tutorial on how to automatically detect checkerboard corners in an image see Auto-detecting checkerboard corners.

Value

an array of three or four dimensions.

Author(s)

Aaron Olsen

See Also

`readLandmarksToArray`, `readLandmarksToList`, `readLandmarksToMatrix`

Examples

```
## GET FILE DIRECTORY FOR PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## SET NUMBER OF ROWS AND COLUMNS
## THESE ARE THE NUMBER OF INTERNAL CORNERS, NOT THE NUMBER OF SQUARES
nx <- 4
ny <- 3

## SET FILE PATHS
file <- matrix(c(paste0(fdir, "rcta_a", 1:3, "_v1.txt"),
                 paste0(fdir, "rcta_a", 1:3, "_v2.txt")), ncol=2)
```

```
## READ MATRIX OF FILES ##
## REVERSE COLUMNS IN FIRST COLUMN OF FILE MATRIX
## REVERSE ROWS IN ALL FILES
readCheckerboardsToArray(file, nx, ny, col.reverse=c(TRUE, FALSE), row.reverse=TRUE)
```

`readLandmarksToArray` *Reads landmark file(s) into an array*

Description

Reads landmarks from one or more files into an array. The files can be input as a vector or matrix and will be grouped into the returned array according to their input grouping. In every case, the first dimension corresponds to the number of landmarks and the second dimension corresponds to the number of landmark dimensions (2 for 2D landmarks, 3 for 3D landmarks, etc.).

Usage

```
readLandmarksToArray(file, na.omit = FALSE, ...)
```

Arguments

`file`	a vector or matrix of file paths to be read. Each file should contain a single landmark matrix.
`na.omit`	whether landmarks with NA values in any file should be omitted.
`...`	further arguments to be passed to `read.table()`.

Details

This function will read landmark matrices from one or more files and use the row names in each matrix to match up corresponding landmarks into a single array, filling in missing landmarks with `NA`. The landmark files are read by `read.file()` and should thus conform to all requirements of `read.file()`. Arguments for `read.file()` can be passed through `readLandmarksToList()` (e.g. `header`, `row.names`, etc.). If the landmark matrices do not have row names, this function assumes that landmarks in the same rows correspond and the number of rows in each landmark file should be the same.

Since stereo camera setups involve at least two camera views and usually more than one orientation per view, `readLandmarksToArray()` can be used to create a single array with multiple orientations and views of the same or overlapping landmark sets. If a vector of file paths is input, `readLandmarksToArray()` returns a three-dimensional array in which the first two dimensions are the landmark matrices and the last dimension is the index in the input file vector. If a matrix of file paths is input, `readLandmarksToArray()` returns a four-dimensional array in which the first two dimensions are the landmark matrices and the last two dimensions are the indices in the input file matrix (see "Examples").

This function is called by `readCheckerboardsToArray` to read in checkerboard corners with additional arguments available through `readCheckerboardsToArray` for manipulating the corner point order.

Value

a landmark array of three or four dimensions.

Author(s)

Aaron Olsen

See Also

readCheckerboardsToArray, readLandmarksToList, readLandmarksToMatrix

Examples

```
## READING IN LANDMARKS WITH ROW NAMES ##
## GET FILE DIRECTORY FOR PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## SET FILES TO LOAD
file <- paste0(fdir, "lm_3d_even_a", 1:3, ".txt")

## LOAD FILES INTO AN ARRAY
lm.array <- readLandmarksToArray(file=file, header=FALSE, row.names=1)

## VIEW THE FIRST FIVE LANDMARKS
lm.array[1:5, , ]

## LOAD FILES INTO AN ARRAY OMITTING ALL NA
lm.array <- readLandmarksToArray(file=file, header=FALSE, row.names=1, na.omit=TRUE)

## VIEW ARRAY
## NOTE THERE ARE ONLY THREE LANDMARKS SHARED AMONG FILES THAT ARE NOT NA
lm.array


## READING SINGLE VS. VECTOR VS. MATRIX FILE INPUTS ##
## SET FILE PATHS
file <- matrix(c(paste0(fdir, "rcta_a", 1:3, "_v1.txt"),
  paste0(fdir, "rcta_a", 1:3, "_v2.txt")), ncol=2)

## READ A SINGLE FILE PATH
## TREATED AS A VECTOR OF LENGTH ONE
readLandmarksToArray(file=file[1, 1])

## READ A FILE PATH VECTOR
readLandmarksToArray(file=file[, 1])

## READ A FILE PATH MATRIX
readLandmarksToArray(file=file)
```

readLandmarksToList *Reads landmark file(s) into a list*

Description

Reads landmarks from one or more files into a list. This function is useful when dealing with curves (semilandmarks) since curve points can be grouped by curve name for other operations.

Usage

```
readLandmarksToList(file, semilandmark.pattern = "[0-9]+$", ...)
```

Arguments

file	a single landmark file or vector of landmark files to be read. Each file should contain a single landmark matrix with row names.
semilandmark.pattern	a regular expression pattern passed to sub() for identifying and grouping curve points. The default is landmark names ending in one or more numbers.
...	further arguments to be passed to read.table().

Details

This function will read a landmark matrix from one or more files and use the row names in each matrix to match corresponding landmarks into list elements, ordered first by the landmark name and then numbered by the index of the file (in file) from which the landmark was read. Landmark lists are the required input format for dltMatchCurvePoints. Landmark lists are also one of three possible input formats for dltReconstruct and allow for curve points to be easily pulled out of a landmark set for curve fitting.

semilandmark.pattern is a regular expression passed to sub() to identify semilandmarks (curve points). By default, the regular expression "[0-9]+$" identifies row names that end in more than one digit (e.g. 'tomium_R004') as curve points. sub() removes the part of the string identified by semilandmark.pattern in order to group all curve points under one curve name (e.g. 'tomium_R004' would be grouped under 'tomium_R'). Curve grouping can be turned off by setting semilandmark.pattern to "". Once grouped, curve points are sorted only by the numeric portion of their row name (identified by semilandmark.pattern using regexpr). Preceding zeros are not necessary. For example, after sorting, the order of the following curve points would be: tomium_R1, tomium_R02, tomium_R9, tomium_R10. Note that if these were sorted simply by row name, the order would be: tomium_R02, tomium_R1, tomium_R10, tomium_R9. Landmarks missing from one or more files are given the value NULL.

The landmark files are read by read.file() and should thus conform to all requirements of read.file(). Arguments for read.file() can be passed through readLandmarksToList() (e.g. header, row.names, etc.).

Value

a landmark list.

Author(s)

Aaron Olsen

See Also

readLandmarksToArray, readLandmarksToMatrix, readCheckerboardsToArray, dltMatchCurvePoints

Examples

```
## GET FILE DIRECTORY FOR PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## SET FILES TO LOAD - TWO DIFFERENT 3D POINT SETS
file <- paste0(fdir, "lm_3d_even_a", 1:2, ".txt")

## READ LANDMARKS INTO A LIST
lm.list <- readLandmarksToList(file=file, row.names=1)

## CURVE POINTS
## CURVE POINTS ARE ABSENT FROM FIRST POINT SET
lm.list[['tomium_R']]

## LANDMARKS PRESENT IN BOTH POINT SETS
lm.list[['quadrate_jugal_R']]

## LANDMARK MISSING FROM SECOND POINT SET
lm.list[['foramen_magnum_inf']]
```

---

readLandmarksToMatrix *Reads a landmark file or files into a matrix*

Description

Reads landmarks from one or more files into a matrix. A single file or vector of files can be input. If more than one file is input, each matrix will be appended to the previous one with matching landmarks in the same row.

Usage

```
readLandmarksToMatrix(file, na.omit = FALSE, ...)
```

Arguments

file	a single landmark file or vector of landmark files to be read. Each file should contain a single landmark matrix.
na.omit	whether landmarks with NA values in any file should be omitted.
...	further arguments to be passed to read.table().

Details

This function will read a landmark matrix from one or more files and use the row names in each matrix to match corresponding landmarks into a single matrix, filling in missing landmarks with NA. The rows correspond to landmarks and the columns correspond to the number of landmark dimensions (2 for 2D landmarks, 3 for 3D landmarks, etc.). Each landmark matrix is appended as new columns onto the existing matrix. So, if three, 2D landmark files are input the resulting matrix would have six columns.

The landmark files are read by `read.file()` and should thus conform to all requirements of `read.file()`. Arguments for `read.file()` can be passed through `readLandmarksToList()` (e.g. `header`, `row.names`, etc.). All landmark matrices must have row names.

Value

a landmark matrix

Author(s)

Aaron Olsen

See Also

`readLandmarksToList`, `readLandmarksToArray`, `readCheckerboardsToArray`

Examples

```
## GET FILE DIRECTORY FOR PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## SET FILES TO LOAD
file <- paste0(fdir, "lm_2d_a3_v", 1:2, "_wna.txt")

## LOAD FILES INTO A MATRIX
readLandmarksToMatrix(file=file, row.names=1)

## LOAD FILES INTO A MATRIX OMITTING NAS
readLandmarksToMatrix(file=file, row.names=1, na.omit=TRUE)
```

---

readShapes *Reads a StereoMorph shape file*

Description

This function reads digitized shape and scaling data from a StereoMorph shape file or files into a list structure.

Usage

```
readShapes(file, fields=NULL)
```

Arguments

file	A shape file, a vector of shape files or a folder containing shape files to be read.
fields	Objects to be returned from the shape file. If NULL, all objects in the file will be returned.

Details

The digitizeImage function makes it possible to save shape and scaling data into a single shape file. This shape file has an XML-like format that allows the readShapes() function to read multiple object types (including vectors, matrices and lists) into a list structure using a generalized routine. All these objects are saved to a list as several elements. The particular elements in the output list will depend on which objects are present in the file. If the object is not present in the file, a call to that object will return NULL. The contents will also differ if multiple files are input. For instance, if one file is input landmarks.pixel will be a matrix but if multiple files are input it will be an array.

The output of print() on the entire output list is formatted for readability given the potentially large matrices contained within the list.

Value

a list of class "shapes" containing any number of the following elements:

image.name	A vector of image names.
image.id	A vector of image IDs.
scaling	A vector of the scaling (real-world units per pixel) of the image.
scaling.units	A vector of the units of scaling.
ruler.pixel	A vector of the interval of the digitized ruler points in pixels.
ruler.interval	A vector of the interval of the digitized ruler points in real-world units.
checkerboard.nx	A vector of the number of internal corners of a checkerboard pattern along one dimension.
checkerboard.ny	A vector of the number of internal corners of a checkerboard pattern along the other dimension.
square.pixel	A vector of the best-fit checkerboard square size in pixels.
square.size	A vector of the best-fit checkerboard square size in real-world units.
landmarks.pixel	A matrix or array of landmark coordinates in pixels.
landmarks.scaled	A matrix or array of scaled landmark coordinates.
ruler.points	A matrix or array of ruler points in pixels.
checker.pixel	A matrix or array of checkerboard points in pixels.
curves.control	A list of Bezier curve control points in pixels.
curves.pixel	A list of Bezier curve points in pixels.
curves.scaled	A list of scaled Bezier curve points.

If any of the above objects are absent from the shape file they will be NULL.

Author(s)

Aaron Olsen

References

See StereoMorph Digitizing App for a tutorial on digitizing and reading shape data.

See Also

digitizeImage

`reflectMissingLandmarks`

Reflects missing landmarks across the plane of symmetry

Description

This function reflects missing bilateral landmarks across the plane of symmetry, optionally averaging left and right landmarks.

Usage

```
reflectMissingLandmarks(lm.matrix, left = '(_L|_l|_left|_LEFT)([0-9]*$)',
                        right = '(_R|_r|_right|_RIGHT)([0-9]*$)',
                        left.remove = '\\2', right.remove = '\\2',
                        left.replace = '_R\\2', right.replace = '_L\\2',
                        average = FALSE)

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

Arguments

`lm.matrix`	a 2D or 3D matrix with landmark names as row names.
`left`	a regular expression to identify left landmarks in the row names of `lm.matrix`.
`right`	a regular expression to identify right landmarks in the row names of `lm.matrix`.
`left.remove`	an expression for input to the `gsub()` function indicating which element of `left` in parentheses should be removed to create a landmark name that is not side-specific (see "Details").
`right.remove`	an expression for input to the `gsub()` function indicating which element of `right` in parentheses should be removed to create a landmark name that is not side-specific (see "Details").
`left.replace`	an expression for input to the `gsub()` function indicating a replacement string for `left` that will turn a left landmark name into a right landmark name (see "Details").

right.replace	an expression for input to the gsub() function indicating a replacement string for right that will turn a right landmark name into a left landmark name (see "Details").
average	a logical indicating whether bilateral landmarks should be averaged.
object	a list of class "reflectMissingLandmarks" (output of this function).
...	further arguments passed to or from other methods.

Details

Currently, the function only accepts left/right designations by matching a regular expression to the row names of lm.matrix. This is preferable since it allows for easier match up between bilateral landmarks. The default regular expression identifies left landmarks by a name ending in "_L", "_l", "_left" or "_LEFT", optionally followed by numbers. For example, "hamulus_left", "hamulus_L" and "zymgomatic_arch_l012" would all be identified as landmarks on the left side. Similarly, "hamulus_right", "hamulus_R" and "zymgomatic_arch_r012" would all be identified as landmarks on the right side. Landmarks not identified as left or right are assumed to fall on the midline.

In order to find corresponding left and right landmarks, the function requires the left.remove and right.remove arguments. The left.remove and right.remove arguments are passed to the base function gsub() as the replacement argument. This is used to generate a landmark name that is not side-specific. For example, "hamulus_left" and "zymgomatic_arch_l012" would become "hamulus" and "zymgomatic_arch012". These will be reverted to their original names at return.

If only a left or right landmark is present in lm.matrix, reflectMissingLandmarks() will create new a new row in lm.matrix for the missing, contralateral landmark. Thus, the output matrix could be longer than the input matrix. The arguments left.replace and right.replace are used to create these new rownames by converting landmark names from left to right or vice versa. By default, the function replaces the existing side designation with "_L" and "_R". For instance, "hamulus_left" and "zymgomatic_arch_L012" would become "hamulus_R" and "zymgomatic_arch_R012", respectively. None of the names of existing landmarks will be modified. Users wanting a different left/right scheme can either change the left.replace and right.replace arguments or make sure that all the bilateral landmarks in lm.matrix are represented by both a left and right landmark (missing values being NA). In this case, left.replace and right.replace will be ignored and no new landmark names will be created.

Once corresponding right and left landmarks have been identified, the plane of object symmetry is found as described by Klingenberg et al. (2002). This includes creating two landmark sets, reflecting one set across the xy-plane, swapping left and right landmark names in one set and performing Procrustes alignment on the two sets. The user then has the option of averaging across the plane of symmetry. This will cause all bilateral landmarks to be mirror images across the midline plane and midline landmarks to lie directly in the midline plane. The input orientation of lm.matrix is maintained. So if average is FALSE, landmarks that were not missing will be unchanged at output (the new landmarks having been filled in around them). If average is TRUE, the positions of the non-missing landmarks will have changed due to averaging but will only be shifted slightly from the original position.

reflectMissingLandmarks() returns an alignment error vector. This is the error (distance) between a left or right landmark and its contralateral landmark (if present) when reflected across the midline plane. This is equivalent to the Procrustes alignment error.

Users with landmark names in alternative formats might find it easier to simply add '_L' and '_R' to the end of left and right landmark names, respectively, rather than re-specifying the regular expression arguments.

For a step-by-step tutorial on how to use `reflectMissingLandmarks()` see Unifying, reflecting and aligning landmarks.

Value

a list of class `"reflectMissingLandmarks"` with the following elements:

`lm.matrix`	a 2D or 3D matrix of landmarks with missing landmarks reflected. This matrix could be longer than the input landmark matrix.
`align.error`	a vector of the error (distance) between between a left or right landmark and its contralateral landmark (if present) when reflected across the plane of symmetry.

Note

This function was modified by A Olsen from the R function `OSymm()` written by A Haber.

Author(s)

Annat Haber, Aaron Olsen

References

Klingenberg, C.P., Barluengua, M., Meyer, A. (2002) Shape analysis of symmetric structures: Quantifying variation among individuals and asymmetry. *Evolution*, **56** (10), 1909–1920.

See Also

`readLandmarksToMatrix`, `alignLandmarksToMidline`

Examples

```
## FIND THE FILE DIRECTORY FOR EXTRA R PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## GET LANDMARKS
file <- paste0(fdir, "lm_3d_unify.txt")

## LOAD FILES INTO A MATRIX
lm.matrix <- readLandmarksToMatrix(file=file, row.names=1)

## ALIGN TO MIDLINE
reflect_missing <- reflectMissingLandmarks(lm.matrix=lm.matrix, average=TRUE)

## PRINT SUMMARY OF ERRORS
print(summary(reflect_missing))
```

`resampleGridImagePoints` *Resamples imaged grid points*

Description

This function fits a 12-parameter image perspective model to imaged grid points and uses the model parameters to produce a grid with the same transformations but consisting of fewer points, effectively "resampling" the number of grid points. In this way, fewer points (but representing the same amount of information) can be used in more computationally intensive steps such as camera calibration. This function is called by `dltCalibrateCameras`.

Usage

```
resampleGridImagePoints(pts, nx, rx, ry, fit.min.break=1,
                        print.progress = FALSE)
```

Arguments

`pts`	a matrix of grid points from an image, such as the internal corners of a checkerboard image.
`nx`	the number of points along the first dimension (e.g. this would be the number of points in each row if points in `pts` are listed first by row).
`rx`	the re-sampled number of points along the first dimension (corresponding to `nx`).
`ry`	the re-sampled number of points along the second dimension (e.g. if `nx` is the number of points per row, this is the new number of points per column).
`fit.min.break`	a minimum returned by `nlminb()` at which `resampleGridImagePoints()` will stop iterating to find a better fit.
`print.progress`	whether the model fit error should be printed. Error is in the same units as `pts`.

Value

a list with the following elements:

`pts`	a matrix of resampled grid points.
`error`	the error (in the same units as `pts`) between the input `pts` and the model fit grid points of the same dimensions.

Author(s)

Aaron Olsen

See Also

`imagePlaneGridTransform`, `readCheckerboardsToArray`, `imagePlaneGridTransformError`, `dltCalibrateCameras`

Examples

```
## FIND THE FILE DIRECTORY FOR EXTRA R PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## GET GRID POINTS
file <- paste0(fdir, "cal_a1_v2.txt")

## SET NUMBER OF GRID ROWS AND COLUMNS
nx <- 21
ny <- 14

## READ THE GRID POINTS INTO A MATRIX
## OUTPUT OF FUNCTION IS AN ARRAY SO WE TAKE THE FIRST ENTRY TO GET MATRIX
coor.2d <- as.matrix(read.table(file))

## RESAMPLE THE GRID WITH THE SAME NUMBER OF POINTS AS IN ORIGINAL
coor_2d_same <- resampleGridImagePoints(pts=coor.2d, nx=nx, rx=21, ry=14,
 print.progress=TRUE)

## RESAMPLE THE GRID WITH A REDUCED NUMBER OF POINTS (4 X 4)
coor_2d_red <- resampleGridImagePoints(pts=coor.2d, nx=nx, rx=4, ry=4,
 fit.min.break=1, print.progress=TRUE)

## PLOT THE ORIGINAL IMAGED POINTS
plot(coor.2d)

## PLOT THE MODELED GRID POINTS WITHIN THE ORIGINAL POINTS
## THE MODEL GOODNESS-OF-FIT CAN BE EVALUATED VISUALLY
points(coor_2d_same$pts, col='red', cex=0.75)

## PLOT THE REDUCED NUMBER OF GRID POINTS
points(coor_2d_red$pts, col='green', lwd=2, cex=1.25)

## PLOT A HISTOGRAM OF THE FIT ERROR
## HERE UNITS ARE PIXELS - MOST POINTS ARE FIT WITHIN 2 PIXELS
hist(coor_2d_same$error)
```

---

rotationMatrixZYX *Returns a matrix to rotate points along the z-, y- and x-axes*

Description

This function returns a rotation matrix that rotates a three-column point matrix about the z-, y- and x-axes, in that order. The three angles of rotation can be specified by a single vector of length three or three separate parameters.

Usage

```
rotationMatrixZYX(t, t2 = NULL, t3 = NULL)
```

Arguments

t	angle (in radians) to rotate around the z-axis or a vector of three angles to rotate around the z-, y- and x-axes, in that order.
t2	if t is a single numeric, this is the angle (in radians) to rotate around the y-axis.
t3	if t is a single numeric, this is the angle (in radians) to rotate around the x-axis.

Value

a 3x3 rotation matrix.

Author(s)

Aaron Olsen

References

http://en.wikipedia.org/wiki/Rotation_matrix#In_three_dimensions

See Also

transformPlanarCalibrationCoordinates

Examples

```
## SPECIFY 3D POINT SET
m <- matrix(c(0,0,0, 1,2,1, 3,0,3, -2,4,1), nrow=4, ncol=3, byrow=TRUE)

## ROTATE 180 DEGREES ABOUT THE Z AXIS
## X AND Y VALUES ARE OPPOSITE AND Z VALUES UNCHANGED
m %*% rotationMatrixZYX(pi, 0, 0)

## ROTATE 180 DEGREES ABOUT THE X AXIS
## Y AND Z VALUES ARE OPPOSITE AND X VALUES UNCHANGED
m %*% rotationMatrixZYX(0, 0, pi)

## ROTATE 90 DEGREES ABOUT THE Z-, THEN Y-, THEN X-AXIS
m %*% rotationMatrixZYX(pi/2, pi/2, pi/2)

## ROTATE 90 DEGREES ABOUT THE Z-, THEN Y-, THEN X-AXIS
m %*% rotationMatrixZYX(c(pi/2, pi/2, pi/2))
```

`transformPlanarCalibrationCoordinates` *Performs rotational and translational transformations to a planar grid*

Description

This function rotates and translates a planar grid or grids according to specified transformation parameters. This function is called by `dltCalibrateCameras`, to find the optimal transformation parameters for a set of arbitrarily oriented grid points that minimizes DLT calibration error. This function is also called by `dltTestCalibration` to generate an ideal grid for accuracy testing.

Usage

```
transformPlanarCalibrationCoordinates(tpar, nx, ny, sx, sy = NULL)
```

Arguments

`tpar`	a vector of six transformation parameters per grid. The first three being rotational parameters (rotation about the z, y and x axes, respectively) and the second three being translational parameters (translation along the x, y and z axes, respectively). For more than one grid, these six values are concatenated as a vector.
`nx`	the number of points along the first dimension (e.g. this would be the number of points in each row if points are listed first by row).
`ny`	the number of points along the second dimension (e.g. this would be the number of points in each column if points are listed first by row).
`sx`	a scaling factor along the first dimension.
`sy`	a scaling factor along the second dimension. If the grid blocks are squares, this can be left as `NULL` and only `sx` will be used.

Value

a matrix of transformed 3D grid coordinates

Author(s)

Aaron Olsen

See Also

`dltCalibrateCameras`, `dltTransformationParameterRMSError`, `dltTestCalibration`

`unifyLandmarks` *Optimally align a set of partial landmark sets*

Description

This function aligns two or more landmark sets using shared points. Corresponding landmarks are identified by matching row names. The function selects a sequence of alignments that minimizes the step-wise alignment error.

Usage

```
unifyLandmarks(lm.array, min.common = dim(lm.array)[2])

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

Arguments

`lm.array`	an array of 2D or 3D landmark matrices. These can be read in from a file or files using `readLandmarksToArray`.
`min.common`	a minimum number of landmarks to use in the alignment. Must be greater than `dim(lm.array)[2]`.
`object`	a list of class `"unifyLandmarks"` (the output of `unifyLandmarks()`).
`...`	further arguments passed to or from other methods.

Details

The input `lm.array` should be an array of 2D or 3D landmark matrices with row names, such as created by `readLandmarksToArray`. The first two dimensions of `lm.array` correspond to the rows and columns of each matrix, respectively. The last dimension of `lm.array` corresponds to each separate landmark matrix.

`unifyLandmarks()` first aligns all pair combinations of landmark sets that share the minimum number of points specified by `min.common`. The two sets that align with the lowest root-mean-square (RMS) error are aligned and the mean positions of all points saved. If there are additional landmark sets, `unifyLandmarks()` aligns each of these with the combined matrix, again identifying the set that aligns with the least RMS error. The alignment with the least error is saved as the new combined landmark matrix. This is repeated for each remaining landmark set, sequentially aligning remaining landmark sets to the combined landmark matrix.

To align two 2D landmark sets, the sets must share at least two landmarks and to align two 3D landmark sets, the sets must share at least three landmarks. These are the default minimum number of points for alignment. A greater number of common points can be specified using the `min.common` parameter. Additionally, in the 3D case, these landmarks must not be collinear. If `lm.array` contains more than two landmark matrices, it is not necessarily required that each landmark set share these minimum number of points with every other landmark set. For example, it may be that two landmark sets do not each separately share the minimum number of landmarks required for alignment with a third landmark set. But if these two landmark sets are combined, they

may then share the required number of landmarks with the third set. During each alignment step, `unifyLandmarks()` skips pairs of matrices that do not share the required number of landmarks. As long as there is some combination of alignments that provide a sufficient number of shared landmarks, all landmark sets can be combined into a single matrix.

If an array consisting of only one landmark matrix is input, the matrix is returned without an alignment operation.

For a step-by-step tutorial on how to use `unifyLandmarks()` see Unifying, reflecting and aligning landmarks.

Value

a list of class `"unifyLandmarks"` with the following elements:

`lm.matrix`	a 2D or 3D landmark matrix.
`unify.seq`	a vector of the order in which landmark sets were aligned.
`unify.error`	a matrix of the alignment error for each shared landmark for each alignment (the number of sets minus one).
`unify.rmse`	a vector of the root-mean-square error of each alignment (the number of sets minus one).

Note

This function was modified by A Olsen from the R function `unifyVD()` written by A Haber.

Author(s)

Annat Haber, Aaron Olsen

References

Rohlf, F.J. (1990) "Chapter 10. Rotational fit (Procrustes) Methods." *Proceedings of the Michigan Morphometrics Workshop*. Ed. F. James Rohlf and Fred L. Bookstein. The University of Michigan Museum of Zoology, 1990. 227–236. Info page at lib.umich.edu

See Also

`findOptimalPointAlignment`

Examples

```
## FIND THE FILE DIRECTORY FOR EXTRA R PACKAGE FILES
fdir <- paste0(path.package("StereoMorph"), "/extdata/")

## SET LANDMARK FILES
file <- paste0(fdir, "lm_3d_even_a", 1:3, ".txt")

## READ LANDMARKS INTO ARRAY
lm.array <- readLandmarksToArray(file, row.names=1)
```

```
## UNIFY LANDMARKS
unify_lm <- unifyLandmarks(lm.array)

## PRINT UNIFICATION SUMMARY
print(summary(unify_lm))
```

Index