Package ‘phenocopter’

April 4, 2012

Type Package

Title utilities for phenocopter processing

Version 1.2.1

Date 2012-02-16

Author Amy Chan

Maintainer Amy Chan <Amy.Chan@csiro.au>

Description functions for parsing pheno log and doing extraction step.

License N/A

Suggests sp,mgcv

Depends grid,imageUtilities,utilitiesR,lensDistortion

Collate
'defaultOptionsPhenocopter.r''drawOrientedPhenocopterImage.r''readLog_ricoh.r''drawCheckImage.r''calculateProjecti

R topics documented:

applyProjectiveTransform

apply a homography/projective transform to some x,y coordinates.

Description

apply a homography/projective transform to some x,y coordinates.

Usage

```
applyProjectiveTransform(transf, x, y = NULL)

applyHomography(transf, x, y = NULL)
```

Arguments

transf	the 3x3 matrix (a,b,c;d,e,f;g,h,1) that contains the coefficients of the transform (see projectiveTransform).
x	1 by n vector of x coordinates, or an nx2 matrix with x in the first row and y in the second (similar to functions like plot
y	NULL if the nx2 x parameter has been supplied already, or a 1 by n vector of y coordinates.

Details

applyHomography is a synonym for this function.

See projectiveTransform for the full information of the transform, but the transformation is applied as follows:

$$\begin{pmatrix} l \\ m \\ n \end{pmatrix} := \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Followed by:

$$u = l/n$$
$$v = m/n$$

Value

an nx2 matrix containing the transformed coordinates.

See Also

Other homography: calculateHomography, calculateProjectiveTransform, projectImage, projectImageFromPoints, projectiveTransform

Other homography: calculateHomography, calculateProjectiveTransform, projectImage, projectImageFromPoints, projectiveTransform

---

`calculateProjectiveTransform`
Calculates homography from an arbitrary quadrilateral to a rectangle.

Description

Calculates homography from an arbitrary quadrilateral to a rectangle.

Usage

```
calculateProjectiveTransform(pts_from, pts_to,
  sorted = FALSE)

calculateHomography(pts_from, pts_to, sorted = FALSE)
```

Arguments

`pts_from`	a 4x2 matrix of points in the source image (X,Y).
`pts_to`	a 4x2 matrix of points in the target image/coordinate system (U,V).
`sorted`	whether the points in `pts_from` correspond **in order** to `pts_to`, or whether the correspondence between the points must be calculated (roughly, matching up "top-left" corners together, "top-right" corners together, etc).

Details

See `projectiveTransform`. This function also be accessed via `calculateHomography`.

Value

a 3x3 matrix with coefficients `(a,b,c;d,e,f;g,h,1)`, in row order.

See Also

Other homography: `applyHomography`, `applyProjectiveTransform`, `projectImage`, `projectImageFromPoints`, `projectiveTransform`

Other homography: `applyHomography`, `applyProjectiveTransform`, `projectImage`, `projectImageFromPoints`, `projectiveTransform`

computePointsForRectangle

compute some `points_to` *for the special case of a rectangle.*

Description

This function takes in `pts_from` and an aspect ratio, and calculates vertices of a rectangle based off these such that the longest side length is preserved.

Usage

```
computePointsForRectangle(pts_from, aspect.ratio,
  origin = c(1, 1))
```

Arguments

`pts_from`	an array of vertices of a quadrilateral, 4 by 2.
`aspect.ratio`	desired aspect ratio of HORIZONTAL dimension to VERTICAL in the output
`origin`	the origin of the desired output rectangle (default (1,1), ie one based coordinates).

Details

The method:

1. calculate all side lengths of the quad defined by `pts_from`
2. pick the longest side length and preserve it
3. use the `aspect.ratio` to determine the other side length
4. generate vertices of the rectangle starting at `origin` with these side lengths.

Value

a 4x2 array with the coordinates of the vertices of the rectangle.

defaultOptionsPhenocopter

Initialise a set of default options for the phenocopter.

Description

Initialise a set of default options for the phenocopter.

Usage

```
defaultOptionsPhenocopter()
```

Details

The options that can be fed in are as follows. The main ones that users may wish to edit are listed first:

Value

a LIST with the default options (see Details).

Commonly-used Parameters

`SAVE_CHECK_IMAGE` boolean, whether to save the check image, intended for the user to verify that the chopping-and-identifying process worked as they expected.

It consists of the input image with the region the user selected drawn on, and also the corresponding individual plots that were identified based on the user's input. The idea is that the user may flick through a directory of these check images as a quick way to verify that no errors were made.

`CHECK_SAVE_FORMAT` The name of the check image (e.g., `%f_check.jpg` saves as `[input filename]_check.jpg`. See the "save formats" section for further details.

`OVERWRITE_IF_EXISTS` Default TRUE, whether to overwrite existing files.

`BASE_SAVE_DIR` The root of the save path. Directory structure will be created within here corresponding to `BASE_SAVE_DIR/site_season/date/flight/modality`.

`ONLY_PROCESS_LOG_IMAGES` Whether to only process images that have a corresponding log file (default True).

`MAX_OUTPUT_WIDTH` Number of pixels wide to make the check image and overlay image (at maximum) - this prevents huge 5000px images. Default 1000. (TODO:rectanglified & barrel images too?)

Lens Distortion Parameters

These parameters are to do with rectifying barrel distortion in the image. For further details on exactly what the parameters mean, see the lensDistortion package (`?lensDistortion`).

`DEFAULT_CALIBRATION_MODEL` One of 'hugin' or 'opencv', which calibration model to use. 'OpenCV' is always used for thermal images. For the differences between the models see the `lensDistortion` pacakage's help files.

Parameters for 'hugin' model of lens distortion, i.e.

$$r_{distorted} = r(d + cr + br^2 + ar^3),$$

or

$$x_{distorted} = x(d + cr + br^a r^3)$$

and

$$y_{distorted} = y(d + cr + br^a r^3)$$

.

`BARREL_DISTORT_{A,B,C,D}` a, b, c and d in the equation above. Default $(0, 0, 0, 1)$, meaning no correction will be applied.

Parameters for the 'opencv' model of lens distortion, i.e.

$$x_{distorted} = kx + 2p_1xy + p2(r^2 + 2x^2)$$

$$y_{distorted} = ky + 2p_2xy + p1(r^2 + 2y^2)$$

where

$$k = 1 + k_1r^2 + k_2r^4 + k_3r^6.$$

`OPENCV_K1`,`OPENCV_K2`,`OPENCV_K3` Radial distortion coefficients. Set all equal to 0 to ignore.

`OPENCV_P2`,`OPENCV_P3` Tangential distortion coefficients, set eqeual to 0 to ignore.

`OPENCV_CX`,`OPENCV_CY` ?? (opencv parameters for recentering image before distortion)

`OPENCV_FX`,`OPENCV_FY` ?? (opencv scaling factor before distortion)

`OPENCV_CXSCALE`,`OPENCV_CYSCALE` ?? (opencv parameters for recentering image after distortion)

`OPENCV_FXSCALE`,`OPENCV_FYSCALE` ?? (opencv scaling factor after distortion)

Output Parameters

`SAVE_BARREL_CORRECTED` Whether to save the input image with the barrel distortion correction applied. Default FALSE.

`SAVE_RECTANGLIFIED` Whether to save the quadrilateral selected from the input image, with the barrel and perspective distortion applied. Default FALSE.

`SAVE_OVERLAY` Whether to save the image of the extracted region (with barrel/perspective correction applied), and lines drawn on top showing the division of the individual plots. Default FALSE.

`SAVE_PLOTS` Whether to save an image of each individual plot. Default FALSE.

`SAVE_RECONS` Whether to save an image consisting of the individual plots pasted back together (with a small border in between). Default FALSE.

`SAVE_TRIMMED_PLOTS` Whether to save images of each individual plot, trimmed on each side. Default FALSE.

`SAVE_GRID` Whether to save the image of the grid lines (useless!). Default FALSE.

How to save each image: use '%f' for the input file name (without the extension), '%H' for the name of the horizontal dimension (e.g. 'row'), '%h' for the value of the horizontal dimension (e.g. 1), '%V' for the name of the vertical dimension (e.g. 'column'), and '%v' for the value of the vertical dimension. '%[HhVv]' *only* apply for the individual plots (ie `PLOTS_SAVE_FORMAT` and `TRIMMED_PLOTS_SAVE_FORMAT`.

Note: all paths are given relative to `BASE_SAVE_DIR/site_season/date/flight/modality`.

`BARREL_CORRECTED_SAVE_FORMAT` Default `%f/%f_barrel.jpg`.

`RECTANGLIFIED_SAVE_FORMAT` Default `%f/%f_barrel_rectanglified.jpg`.

`OVERLAY_SAVE_FORMAT` Default `%f/%f_overlay.jpg`.

`GRID_SAVE_FORMAT` Default `%f/%f_grid.jpg`.

`PLOTS_SAVE_FORMAT` Default `%f/%f_%H%h-%V%v.jpg`, e.g. 'img1234/img1234_Row3-Plot5.jpg'.

`TRIMMED_PLOTS_SAVE_FORMAT` Default `%f/%f_%H%h-%V%v_trimmed.jpg`, e.g. 'img1234/img1234_Row3-Plot5_trimmed.jpg'.

`RECONS_SAVE_FORMAT` Default `%f/%f_reconstructed.jpg`.

Cosmetic parameters

TRIM_SIDES ** GOING TO BE DEPRECIATED, IGNORE **. Default TRUE.

TRIM_PERCENT How much to trim the sides of each individual plot, if one is saving the reconstructed image, check image, or individual trimmed plots. ** MAY BE REMOVED **. Default c(0.1,0.1,0,0) (left,right,top,bottom).

OVERLAY_LINE_WIDTH Width of lines surrounding the selected region in the check image. Default 2.

OVERLAY_LINE_COLOUR Colour of lines denoting the selected region in the check image. Default 'red'.

Experimental parameters

These may or may not be implemented.

SAVE_PARTIAL_PLOTS,PARTIAL_PLOTS_SAVE_FORMAT Whether to save plots that are not wholly in the image, and how to save them.

Parameters you usually would not need to modify:

SAVE_IN_COLOUR Whether to save thermal images in colour. Default TRUE.

CAMERA_IMAGE_FORMAT The extension (case-insensitive) for camera images. Default "jpg". (may be phased out soon).

SAVE_EXPERIMENT Whether to save details that allow the analysis to be repeated at a later date. DO NOT CHANGE THIS!. Default TRUE.

EXPERIMENT_SAVE_FORMAT How to save the details that allow the analysis to be repeated, for the moment "%f.rda". DO NOT CHANGE FOR NOW

See Also

Other phenocopter: loadOptions, RUN_PHENOCOPTER_EXPERIMENT, SETUP_PHENOCOPTER_EXPERIMENT

---

drawCheckImage *given pheno image and corresponding reconstructed image, draw both in a check image. *GRID GRAPHICS**

Description

given pheno image and corresponding reconstructed image, draw both in a check image. *GRID GRAPHICS*

Usage

```
drawCheckImage(img, patchImage, filename = NULL,
  flip = FALSE, maxWidth = 1280, dev.off = NULL, ...)
```

Arguments

`img`	the phenocopter image
`patchImage`	the reconstructed image
`filename`	path to save output as
`flip`	whether to flip images
`dev.off`	boolean, whether to turn the device off after plotting
`maxWidth`	the maximum width the image should be. Can be either a number (in which case it will be interpreted as *pixels*), or a unit of the form `unit(width,type)` where `type` is one of 'npc' or 'inches' (see `convertScreenUnits`). The image will be downsampled to this width in the display, but the *native* coordinates will still be as for the full-size image.
`...`	fed into `drawOrientedPhenocopterImage`

Value

a list of viewports:

root the root viewport

left the left viewport (native coordinates are pixels), name 'vp.left' (pheno image)

right the right viewport (native coordinates are pixels), name 'vp.right' (reconstructed image)

See Also

`convertScreenUnits`

Other draw: `drawOrientedPhenocopterImage`

`drawOrientedPhenocopterImage`

given pheno image and log info, flips it so that they all face the same way, draws arrows on, etc.

Description

given pheno image and log info, flips it so that they all face the same way, draws arrows on, etc.

Usage

```
drawOrientedPhenocopterImage(imgOrPath, flip,
  PHENO_COLOUR = "yellow", ORIGIN_COLOUR = "white", ...)
```

Arguments

`imgOrPath`	the image to draw on
`flip`	boolean, flip the image upside down?
`ORIGIN_COLOUR`	colour to draw the arrow pointing to the start ('white'). NA to omit origin decorations.
`PHENO_COLOUR`	colour to draw the arrow pointing in pheno direction ('yellow'). NA to omit pheno decorations.
`...`	passed into `drawImage`

Note

requires grid & imageUtilities

See Also

Other draw: `drawCheckImage`

`generatePolyGrid` *generate a grid of points that lie in a convex polygon.*

Description

generate a grid of points that lie in a convex polygon.

Usage

```
generatePolyGrid(xrange, yrange, poly, by = 1,
  quietly = FALSE)
```

Arguments

`xrange`	`c(start,end)` of the x grid to generate
`yrange`	`c(start,end)` of the y grid to generate
`poly`	an nx2 matrix, each row being an (X,Y) for each n vertices of the polygon.
`by`	Amount to increment the grid by.
`quietly`	boolean, whether to give a warning if none of the packages could be loaded (in which case the input grid is returned, unfiltered).

Details

requires sp or mgcv packages. If none can be found, it just returns the input points unfiltered.

Value

an mx2 matrix of all pairs of `seq(xrange[1],xrange[2], by=by)` with `seq(yrange[1],yrange[2],by=by)`, filtered to only include (x,y) pairs that fall inside the polygon.

`loadOptions` *loads all parameters necessary to re-run a phenocopter experiment.*

Description

loads all parameters necessary to re-run a phenocopter experiment.

Usage

```
loadOptions(fileList, outputDir,
  fmt = defaultOptionsPhenocopter()$EXPERIMENT_SAVE_FORMAT)
```

Arguments

`fileList`	list of *image* files we want to re-process
`outputDir`	path to the *root* output directory where the previous analyses were done. Since output paths look like `rootpath/siteseason/date/flight/modality`, `outputDir` corresponds to `rootpath`.
`fmt`	the format of the saved files (for example "%f.rda"), you shouldn't need this parameter unless you messed with it earlier (I hope you didn't...)

Value

a `list(options,details)` containing the amalgamated options and details to feed into `RUN_PHENOCOPTER_EXPERIMENT`.

Note

If the given images have conflicting `options`, then a warning is raised and *only the options of the first file are used* (for conflicts).

See Also

Other phenocopter: `defaultOptionsPhenocopter`, `RUN_PHENOCOPTER_EXPERIMENT`, `SETUP_PHENOCOPTER_EXPERIMENT`

Examples

```
## Not run:
# suppose we'd previously done analyses on 'vis/img1.jpg' and
# 'thermal/img2.bin', which live in
# '/data/gatton_summer/28-11-11/flight01/'.
# Furthermore the analysis results are housed under '/results'.
baseDir <- '/data/gatton_summer/28-11-11/flight01'
obj <- loadOptions( c( file.path(baseDir,'vis','img1.jpg'),
                file.path(baseDir,'thermal','img2.bin') ),
             outputDir='/results' )
# this looks in '/results/gatton_summer/28-11-11/flight-1/vis/img1.rda'
# (and so on) to load that image's data from the .rda.
# These are all amalgamated into obj$opts and obj$details.
# I could modify some options if I liked, say
obj$options$SAVE_CHECK_IMAGE <- TRUE

# To re-run the processing, I'd then do:
RUN_PHENOCOPTER_EXPERIMENT( obj$details, obj$options )

## End(Not run)
```

---

parsePath *parse an image path in the required format and return information*

Description

This parses the path of an image in the format required for the phenocopter project (`path/to/siteseason/date/flight/moc` and returns the modality, flight, date, siteseason, and root.

Usage

```
parsePath(pth, hasModality)
```

Arguments

`pth`	the path to parse
`hasModality`	whether the path includes the modality, or is just to the flight.

Details

The phenocopter project *requires* that images are saved in the following directory structure: `root/siteseason/date/flight/modality`, where:

root is the path to where *all* the images across the entire project will be housed

siteseason is the site and growing season (e.g. `gatton_summer`)

date is the date of the flight(s)

flight one folder per flight, with takeoff time

modality one folder per imaging modality - 'vis', 'thermal', or 'nir'

filename the name of the file

Value

a list with components corresponding to siteseasonn, date, flight and modality (see Details). or NULL if the path could not be parsed.

Note

if the path ends with '/' it is assumed that the filename is absent.

Examples

```
obj <- parsePath('/data/crop/phenocopter/gatton_summer/28-11-11/flight01_1423/vis/img1.jpg',
hasModality=TRUE)
obj$modality   # vis
obj$flight     # flight01_1423
obj$date       # 28-11-11
obj$siteseason # gatton_summer
obj$root       # /data/crop/phenocopter

obj <- parsePath('/data/gatton_summer/28-11-11/flight01_1423/ricoh.log',
                 hasModality=FALSE)
obj$modality   # NA

obj <- parsePath('/data/gatton_summer/28-11-11/flight01_1423/',
                 hasModality=FALSE)
obj$modality   # NA
obj$filename   # NA
```

`projectImage` *Apply a projective transform to an entire image.*

Description

Apply a projective transform to an entire image.

Usage

```
projectImage(img, transf, one.based = TRUE,
  border.value = 0)
```

Arguments

`img`	the image to transform
`transf`	the transform *from* `img` to the output image.
`one.based`	whether transform was calculated from one-based indices (does this make a difference??)
`border.value`	the value to use (in the output image) for pixels in the output image with no corresponding pixels in the input image.

Details

One could generate a grid of points in the input image, transform them, and use them as indices to populate the output image. However, is it possible that the output image would then have many internal holes (for example, if the output image is much larger than the input, then points that are next to each other in the input can become spaced out in the output).

To get around this, we take this approach:

1. Transform the *vertices* of the input image to find the vertices of the output image.
2. Use the transformed vertices to generate a grid of points that lie within this output quadrilateral.
3. Use the *inverse transform* to transform these quad points *back* to indices into the *original* image.
4. Assign output values to input values using the indices.

This makes sure there won't be internal holes in the output image, only (possibly) bits around the border if the output quadrilateral is not exactly rectangular.

Value

an image with the same number of channels as the input, consisting of the transform *applied* to the entire input image. See `applyProjectiveTransform` to see the details of that.

That is, the rectangle that is the input image is mapped to some output quadrilateral via `transf`, and the output image has the same size as the minimum enclosing rectangle for that quadrilateral, and has the quadrilateral pasted in it.

See Also

Other homography: `applyHomography`, `applyProjectiveTransform`, `calculateHomography`, `calculateProjectiveTra`
`projectImageFromPoints`, `projectiveTransform`

`projectImageFromPoints` *Apply a projective transform to an image, retaining just the quadrilateral defining the transform.*

Description

Apply a projective transform to an image, retaining just the quadrilateral defining the transform.

Usage

```
projectImageFromPoints(img, pts_from, pts_to,
  one.based = TRUE, border.value = 0, restrict = TRUE,
  ...)
```

Arguments

`img`	the image to transform
`pts_from`	a 4x2 matrix of points in the source image (X,Y).
`pts_to`	a 4x2 matrix of points in the target image/coordinate system (U,V).
`one.based`	whether pts_from and pts_to are one-based or zero-based.
`border.value`	the value to use (in the output image) for pixels in the output image with no corresponding pixels in the input image.

`restrict`	whether to return *just* the quad defined by pts_to, or the entire image (transformed).
`...`	further arguments to `calculateProjectiveTransform` (e.g. `sorted`).

Details

as usual, this won't work with non-convex quadrilaterals.

Value

an image with the same number of channels as the input, consisting of the region pts_to filled in with values from pts_from (if `restrict=TRUE`). Otherwise, returns the transform applied to the entire input image (same behaviour as `projectImage`.

See Also

Other homography: `applyHomography`, `applyProjectiveTransform`, `calculateHomography`, `calculateProjectiveTra`
`projectImage`, `projectiveTransform`

`projectiveTransform` *Projective transforms*

Description

A projective transform, or homography, is (for our purposes here) an invertible transformation from a 2D plane to another that maps straight lines to straight lines.

Details

Another way to say what a projective transform is, is a mapping of one quadrilateral in a 2D plane to another

(Actually, it is much more than that, but I can't explain it very well).

In particular, it can be used to correct an image for perspective distortion.

For example, if one were to take a picture of a rectangular piece of paper with the camera not being head-on, the outline of the piece of paper *in the photo* would not be a rectangle, i.e. would not have right angles between each side.

A projective transform can be used to map the paper's *outline in the picture* to its actual shape in real life Hence a projective transform can be used to transform an image such that it looks like the camera was directly looking at the subject when the photo was taken.

Functions

The following is a list of functions implemented for projective transforms (note, all functions 'xxxProjectiveTransform' are also aliased to 'xxxHomography for ease of typing):

`calculateProjectiveTransform` Computes the set of coefficients a–h that define the transform.

`applyProjectiveTransform` Applies a set of coefficients describing a projective transform to a set of (x,y) points.

codeprojectImage applies a transform to an entire image.

`projectImageFromPoints` calculates a transform from a given quadrilateral, rectifying *just* that quadrilateral.

`computePointsForRectangle` attempts to calculate appropriate target points (rectangular) for a given set of input points (quadrilateral).

Maths

Let (x, y) be the coords for one quadrilateral and (u, v) for the other. The homography, or projctive transform between the two is as follows:

$$u = (ax + by + c)/(gx + hy + 1)$$

$$v = (dx + ey + f)/(gx + hy + 1)$$

where a, b, c, d, e, f, g and h are unknown constants.

Thus, knowing four points in both (x,y) and (u,v) space gives rise to 8 equations of the above forms (4 of each), being 8 equations in 8 unknowns. Once a to h have been found this gives the required transform.

To simplify matters further, let's designate one vertex of each quadrilateral to be at $(0, 0)$ in that coordinate system, and pick them such that they correspond.

Substituting in the equations above, one sees that this correspondence gives $c == f == 0$.

Using this and multiplying by $(gx + hy + 1)$ on both sides and rearranging, one obtains:

$$u = xa + yb - uxg - uyh$$

$$v = xd + ye - vxg - vyh$$

Rewritten in matrix form:

$$\begin{pmatrix} x & y & 0 & 0 & -ux & -uy \\ 0 & 0 & x & y & -vx & -vy \end{pmatrix} \begin{pmatrix} a \\ b \\ d \\ e \\ g \\ h \end{pmatrix} = \begin{pmatrix} u \\ v \end{pmatrix}$$

Substituting the three other known (x, y) & (u, v) pairs (i.e. besides the origin) allows a, b, d, e, g and h to be solved for.

In particular, one can rewrite this as a giant matrix equation (with the other 3 x,y pairs in) as follows:

$$\begin{pmatrix} x_1 & y_1 & 0 & 0 & -u_1x_1 & -u_1y_1 \\ 0 & 0 & x_1 & y_1 & -v_1x_1 & -v_1y_1 \\ x_2 & y_2 & 0 & 0 & -u_2x_2 & -u_2y_2 \\ 0 & 0 & x_2 & y_2 & -v_2x_2 & -v_2y_2 \\ x_3 & y_3 & 0 & 0 & -u_3x_3 & -u_3y_3 \\ 0 & 0 & x_3 & y_3 & -v_3x_3 & -v_3y_3 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ g \\ h \end{pmatrix} = \begin{pmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \end{pmatrix}$$

In particular, let's now utilise that our (x,y) is a rectangle with horizontal and vertical lengths HL and VL. Suppose the following correspondences are known:

x, y	u, v
$0, 0$	$u_{00}, v_{00} = 0, 0$
$0, VL$	u_{0L}, v_{0L}
$HL, 0$	u_{L0}, v_{L0}
HL, VL	u_{LL}, v_{LL}

Then the big matrix equation reduces to:

$$\begin{pmatrix} 0 & VL & 0 & 0 & 0 & -u_{0L} \\ 0 & 0 & 0 & VL & 0 & -v_{0L} \\ HL & 0 & 0 & 0 & -u_{L0} & 0 \\ 0 & 0 & HL & 0 & -v_{L0} & 0 \\ HL & VL & 0 & 0 & -u_{LL} & -u_{LL} \\ 0 & 0 & HL & VL & -v_{LL} & -v_{LL} \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ g \\ h \end{pmatrix} = \begin{pmatrix} u_{0L} \\ v_{0L} \\ u_{L0} \\ v_{L0} \\ u_{LL} \\ v_{LL} \end{pmatrix}$$

Once the coefficients have been solved for, a convenient way to convert from (x, y) to (u, v) is:

$$\begin{pmatrix} l \\ m \\ n \end{pmatrix} := \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Followed by:

$$u = l/n$$

$$v = m/n$$

See Also

Other homography: `applyHomography`, `applyProjectiveTransform`, `calculateHomography`, `calculateProjectiveTra`
`projectImage`, `projectImageFromPoints`

`readLog_ricoh` *Read a ricoh log.*

Description

reads in a ricoh log.

Usage

```
readLog_ricoh(filename = NULL, imageDir = NULL,
  CAMERA_IMAGE_FORMAT = "JPG")
```

Arguments

`filename`	the path to ricoh.log
`CAMERA_IMAGE_FORMAT`	extension of camera images to look for,
`imageDir`	the directory the images for the log are in case-insensitive. time).

Value

data frame with columns:

time time (seconds) of image acquisition - a system time, not the actual time

lat latitude (degrees)

long longitude (degrees)

height altitude (metres)

roll roll (left-right rotation) (degrees)

pitch pitch (forward-back rotation) (degrees)

yaw yaw (like heading but 0 is not necessary due north) (degrees)

path path to corresponding image

`rownames(ricoh)` will be the image name (not the full path).

If `imageDir` is NULL, the path column won't be there and log lines won't be aligned to images.

RUN_PHENOCOPTER_EXPERIMENT *Workhorse function processing phenocopter images to individual plots.*

Description

Workhorse function processing phenocopter images to individual plots.

Usage

```
RUN_PHENOCOPTER_EXPERIMENT(details,
  options = defaultOptionsPhenocopter())
```

Arguments

`details`	a data frame, being the 'details' component of the list returned by `SETUP_PHENOCOPTER_EXPERIMENT`.
`options`	a list of options for the experiment, see `defaultOptionsPhenocopter`.

Details

This function performs the non-interactive part of the phenocopter image processing workflow.

It takes in a list of images and parameters that the user has specified via `defaultOptionsPhenocopter` and `SETUP_PHENOCOPTER_EXPERIMENT`, and performs the following steps:

1. correct input images for lens distortion using the lensDistortion package functions.
2. extract the (rectangular) region of interest the user specified in `SETUP_PHENOCOPTER_EXPERIMENT` and perform perspective distortion correction on it to turn it into an actual rectangular region.
3. using the plot/row information the user specified earlier, divide the rectangle up into an appropriate number of rows/columns of individual plots, labelling each one.
4. generate a number of output images (depending on what was requested), including a check image (showing the region of interest next to the extracted, labelled plots), pictures of each individual plot, etc.

Value

the same data frame fed in via the `details` argument, with a few extra bits of information inserted (mainly file paths of where all the output images went).

See Also

Other phenocopter: `defaultOptionsPhenocopter`, `loadOptions`, `SETUP_PHENOCOPTER_EXPERIMENT`

`SETUP_PHENOCOPTER_EXPERIMENT` *Function for the user-interactive part of processing phenocopter images.*

Description

Function for the user-interactive part of processing phenocopter images.

Usage

```
SETUP_PHENOCOPTER_EXPERIMENT(options = defaultOptionsPhenocopter(),
  fileList = NULL)
```

Arguments

`options`	a list of options for the experiment, see `defaultOptionsPhenocopter`.
`fileList`	a character vector containing paths to images to be processed. If not supplied a file selection dialogue will be shown allowing you to select these images (possibly quite primitive, depending on your OS/R interface)

Details

This function performs the interactive part of the phenocopter image processing workflow.

It gathers the information that will be used by `RUN_PHENOCOPTER_EXPERIMENT` to actually process the images.

The user will be walked through the following steps:

1. selecting a region of interest from the phenocopter image (a rectangular region containing the plots of interest to the user)
2. using a (.csv) field plan, the user identifies the corner plots of the selected region.
3. specifying the length and width of each individual plot

Note you may be asked to provide a log file for the flight. This is only used so that images taken in the same flight are aligned with each other. For example when the helicopter turns around between laps the resulting images will be upside down with respect to those on the previous lap. The ricoh log is (currently) only used to flip them all to face the same way. It isn't crucial.

Value

a list:

options The input options fed into the function, possibly with a few changes (depending on whether the user decided to change the default save directory within the function, etc...)

details A data frame with many columns describing the information that the user fed in to the algorithm, so that it may be reconstructed later without the user having to go through the process all over again

log A list with keys 'vis', 'thermal', and 'nir', each (possibly) containing a data frame of the ricoh.log for that flight, registered with the image information.

See Also

Other phenocopter: `defaultOptionsPhenocopter`, `loadOptions`, `RUN_PHENOCOPTER_EXPERIMENT`

Index