Package ‘recolorize’

March 3, 2025

Title Color-Based Image Segmentation

Version 0.2.0

Description Automatic, semi-automatic, and manual functions for generating color maps from images. The idea is to simplify the colors of an image according to a metric that is useful for the user, using deterministic methods whenever possible. Many images will be clustered well using the out-of-the-box functions, but the package also includes a toolbox of functions for making manual adjustments (layer merging/isolation, blurring, fitting to provided color clusters or those from another image, etc). Also includes export methods for other color/pattern analysis packages (pavo, patternize, colordistance).

License CC BY 4.0

Encoding UTF-8

LazyData true

RoxygenNote 7.3.1

Imports imager, stats, png, pavo, grDevices, graphics, mgcv, colorRamps, plotfunctions, abind, raster, plot3D

Depends R (>= 4.1.0)

Suggests knitr, rmarkdown, sf, smoothr, clue, spatstat.geom, methods

VignetteBuilder knitr

URL `https://hiweller.github.io/recolorize/`, `https://github.com/hiweller/recolorize`

BugReports `https://github.com/hiweller/recolorize/issues`

NeedsCompilation no

Author Hannah Weller [aut, cre] (<`https://orcid.org/0000-0002-5252-4282`>)

Maintainer Hannah Weller <`hannahiweller@gmail.com`>

Repository CRAN

Date/Publication 2025-03-03 21:00:03 UTC

Contents

`absorbLayer` *Absorb a layer into its surrounding color patches*

Description

Absorb a layer into its surrounding color patches

Usage

```
absorbLayer(
  recolorize_obj,
  layer_idx,
  size_condition = function(s) s <= Inf,
  x_range = c(0, 1),
  y_range = c(0, 1),
  remove_empty_layers = TRUE,
  plotting = TRUE
)
```

Arguments

`recolorize_obj`	A `recolorize` object.
`layer_idx`	The numeric index of the layer to absorb.
`size_condition`	A condition for determining which components to absorb, written as a function. The default (`function(l) l <= Inf`) will affect all components, since they all have fewer than infinity pixels.
`x_range`, `y_range`	The rectangular bounding box (as proportions of the image width and length) for selecting patches. Patches with at least partial overlap are counted. Defaults (0-1) include the entire image. See details.
`remove_empty_layers`	Logical. If the layer is completely absorbed, remove it from the layer indices and renumber the existing patches? (Example: if you completely absorb layer 3, then layer 4 -> 3 and 5 -> 4, and so on).
`plotting`	Logical. Plot results?

Details

This function works by splitting a layer into spatially distinct 'components' using imager::split_connected. A contiguous region of pixels is considered a single component. Only components which satisfy both the `size_condition` and the location condition (specified via `x_range` and `y_range`) are absorbed, so you can be target specific regions with (ideally) a minimum of fuss.

The `size_condition` is passed as a function which must have a logical vector output (`TRUE` and `FALSE`) when applied to a vector of sizes. Usually this will be some combination of greater and less than statements, combined with logical operators like `&` and `|`. For example, `size_condition = function(x) x > 100 | x < 10` would affect components of greater than 100 pixels and fewer than 10 pixels, but not those with 10-100 pixels.

The `x_range` and `y_range` values set the bounding box of a rectangular region as proportions of the image axes, with the origin (0, 0) in the bottom left corner. Any patch which has at least partial overlap with this bounding box will be considered to satisfy the condition. When selecting this region, it can be helpful to plot a grid on the image first to narrow down an approximate region (see examples).

Value

A `recolorize` object.

See Also

editLayers for editing layers using morphological operations; thresholdRecolor for re-fitting the entire image without minor colors.

Examples

```
img <- system.file("extdata/fulgidissima.png", package = "recolorize")

# get an initial fit using recolorize + recluster:
fit1 <- recolorize2(img, bins = 3, cutoff = 65, plotting = FALSE)
# this looks okay, but the brown patch (3) has some speckling
# in the upper right elytron due to reflection, and the orange
# patch (4) has the same issue

# the brown patch is easier to deal with, since size thresholding alone is
# sufficient; we want to leave the stripes intact, so we'll absorb components
# that are 50-250 pixels OR fewer than 20 pixels (to get the tiny speckles),
# leaving the eyes intact
fit2 <- absorbLayer(fit1, layer_idx = 3,
                    size_condition = function(x) x <= 250 &
                      x >= 50 |
                      x < 20)

# what about the orange speckles? this is more difficult, because
# we want to retain the border around the brown stripes, but those patches
# are quite small, so size thresholding won't work

# but we just want to target pixels in that one region, so we can first
```

```
# determine a bounding box for it by plotting a grid:
plotImageArray(constructImage(fit2$pixel_assignments,
                    fit2$centers))
axis(1, line = 3); axis(2, line = 1)
abline(v = seq(0, 1, by = 0.1),
       h = seq(0, 1, by = 0.1),
       col = grey(0.2),
       lty = 2)
# x-axis range: 0.5-0.7
# y-axis range: 0.55-0.75
# let's try it:
fit3 <- absorbLayer(fit2, layer_idx = 4,
                    size_condition = function(x) x < 100,
                    x_range = c(0.5, 0.7),
                    y_range = c(0.55, 0.75))
# looks pretty good
```

`add_image` *Add a raster image to a plot*

Description

Adds a raster image (a 3D array) to an existing plot as an image. A silly, generic function, but nice for visualization. Sort of like graphics::points, but for images.

Usage

```
add_image(obj, x = NULL, y = NULL, width = NULL, interpolate = TRUE, angle = 0)
```

Arguments

`obj`	An array of the dimensions height x width x channels, such as read in by png::readPNG or readImage, or the `original_img` and `recolored_img` elements of a `recolorize` object.
`x, y`	The x and y coordinates on which the image should be centered.
`width`	Image width, in x-axis units.
`interpolate`	Passed to graphics::rasterImage. Use linear interpolation when scaling the image?
`angle`	Passed to graphics::rasterImage. The angle (in degrees) for rotating the image.

Value

Nothing; adds an image to the existing plot window.

Examples

```
images <- dir(system.file("extdata", package = "recolorize"),
              ".png", full.names = TRUE)
x <- runif(5)
y <- runif(5)
plot(x, y,
     xlim = range(x) + c(-0.2, 0.2),
     ylim = range(y) + c(-0.2, 0.2))
for (i in 1:length(images)) {
  img <- readImage(images[i])
  add_image(img, x[i], y[i], width = 0.1)
}
```

`adjust_color` *Adjust the saturation and brightness of a color*

Description

Adjusts the saturation and brightness of RGB colors.

Usage

```
adjust_color(
  rgb_color,
  which_colors = "all",
  saturation = 1,
  brightness = 1,
  plotting = FALSE
)
```

Arguments

`rgb_color`	Matrix of RGB colors (0-1 scale).
`which_colors`	The indices of the colors to change. Can be a numeric vector or "all" to adjust all colors.
`saturation`	Factor by which to multiply saturation. > 1 = more saturated, < 1 = less saturated.
`brightness`	Factor by which to multiply brightness.
`plotting`	Logical. Plot resulting color palettes?

Value

A matrix of adjusted RGB colors.

Examples

```
# generate a palette:
p <- grDevices::palette.colors()

# convert to RGB using col2rgb, then divide by 255 to get it into a
# 0-1 range:
p <- t(col2rgb(p)/ 255 )

# we can adjust the saturation and brightness by the same factor:
p_1 <- adjust_color(p, saturation = 2,
                    brightness = 1.5,
                    plotting = TRUE)

# or we can pass a vector for the factors:
p_2 <- adjust_color(p,
                    saturation = seq(0, 2, length.out = 9),
                    plotting = TRUE)

# or we can target a single color:
p_3 <- adjust_color(p, which_colors = 4,
                    saturation = 2, brightness = 2,
                    plotting = TRUE)
```

`apply_imager_operation` *Apply imager operations to layers of an image*

Description

Internal wrapper function for applying any of several `imager` morphological operations for cleaning pixsets.

Usage

```
apply_imager_operation(pixset, imager_function, ...)
```

Arguments

`pixset`	An object of class `pixset`. Usually a layer from `splitByColor()` that has been converted to a `pixset` object.
`imager_function`	The name of an imager morphological operation that can be performed on a pixset, passed as a string. See details.
`...`	Further arguments passed to the imager function being used.

Details

Current imager operations are:

- `imager::grow()`: Grow a pixset
- `imager::shrink()`: Shrink a pixset
- `imager::fill()`: Remove holes in an pixset. Accomplished by growing and then shrinking a pixset.
- `imager::clean()`: Remove small isolated elements (speckle). Accomplished by shrinking and then growing a pixset.

Value

The resulting pixset after applying the specified morphological operation.

`array_to_cimg` *Converts from a raster array to a cimg object*

Description

What it says it does.

Usage

```
array_to_cimg(x, flatten_alpha = TRUE, bg = "white", rm_alpha = TRUE)
```

Arguments

`x`	An image array, i.e. as read in by readPNG.
`flatten_alpha`	Logical. Flatten the alpha channel?
`bg`	Passed to `imager::flatten.alpha()`. Pixel color for previously transparent pixels.
`rm_alpha`	Logical. Remove the alpha channel? Note this will "reveal" whatever is hidden behind the transparent pixels, rather than turn them white.

Value

A `cimg` object.

`array_to_RasterStack` *Convert from an array to a raster stack*

Description

Convert from an image array to a raster stack, optionally using the alpha channel as a mask.

Usage

```
array_to_RasterStack(
  img_array,
  type = c("stack", "brick"),
  alpha_mask = TRUE,
  return_alpha = FALSE
)
```

Arguments

`img_array`	An RGB array.
`type`	Type of Raster* object to return. One of either "stack" (raster::stack) or "brick" (raster::brick).
`alpha_mask`	Logical. Use the alpha channel as a background mask?
`return_alpha`	Logical. Return the alpha channel as a layer?

Value

A Raster* object, either `RasterStack` or `RasterBrick` depending on the `type` argument.

`assignPixels` *Assign a 2D matrix of pixels to specified colors*

Description

Assign a 2D matrix of pixels to specified colors

Usage

```
assignPixels(
  centers,
  pixel_matrix,
  color_space = "Lab",
  ref_white = "D65",
  adjust_centers = TRUE
)
```

Arguments

`centers`	Matrix of color centers (rows = colors, columns = channels).
`pixel_matrix`	Matrix of pixel colors (rows = pixels, columns = channels).
`color_space`	Color space in which to minimize distances, passed to `[grDevices]{convertColor}`. One of "sRGB", "Lab", "Luv", or "XYZ". Default is "Lab", a perceptually uniform (for humans) color space.
`ref_white`	Reference white for converting to different color spaces. D65 (the default) corresponds to standard daylight.
`adjust_centers`	Logical. Should the returned color clusters be the average value of the pixels assigned to that cluster? See details.

Details

This is a largely internal function called by `imposeColors()` for recoloring an image based on extrinsic colors. If `adjust_centers` = `TRUE`, then after assigning pixels to given color centers, the location of each color center is replaced by the average color of all the pixels assigned to that center.

Value

A list of class `color_clusters`, containing:

1. `pixel_assignments`: The color center assignment for each pixel.
2. `centers`: A matrix of color centers. If `adjust_centers` = `FALSE`, this will be identical to the input of `centers`.
3. `sizes`: The number of pixels assigned to each cluster.

Examples

```
# RGB extremes (white, black, red, green, blue, yellow, magenta, cyan)
ctrs <- matrix(c(1, 1, 1,
                 0, 0, 0,
                 1, 0, 0,
                 0, 1, 0,
                 0, 0, 1,
                 1, 1, 0,
                 1, 0, 1,
                 0, 1, 1), byrow = TRUE, ncol = 3)

# plot it
recolorize::plotColorPalette(ctrs)

# create a pixel matrix of random colors
pixel_matrix <- matrix(runif(3000), ncol = 3)

# assign pixels
reassigned <- recolorize::assignPixels(ctrs, pixel_matrix, adjust_centers = TRUE)
recolorize::plotColorPalette(reassigned$centers)

# if we turn off adjust_centers, the colors remain the same as the inputs:
```

```
keep.centers <- recolorize::assignPixels(ctrs, pixel_matrix, adjust_centers = FALSE)
recolorize::plotColorPalette(keep.centers$centers)
```

`backgroundCondition` *Generate a background condition for masking*

Description

Internal function for parsing potential background conditions. Prioritizes transparency masking if conflicting options are provided. See details.

Usage

```
backgroundCondition(
  lower = NULL,
  upper = NULL,
  center = NULL,
  radius = NULL,
  transparent = NULL,
  alpha_channel = FALSE,
  quietly = TRUE
)
```

Arguments

`lower, upper`	RGB triplet ranges for setting a bounding box of pixels to mask.
`center, radius`	RGB triplet and radius (as a proportion) for masking pixels within a spherical range.
`transparent`	Logical or `NULL`. Use transparency to mask? Requires an alpha channel.
`alpha_channel`	Logical. Is there an alpha channel?
`quietly`	Logical. Print a message about background masking parameters?

Details

Prioritizes transparency. If `transparency = TRUE` but other options (such as `lower` and `upper`) are specified, then only transparent pixels will be masked. If `transparency = TRUE` but there is no alpha channel (as in a JPEG image), this flag is ignored and other options (`lower` and `upper` or `center` and `radius`) are used instead.

This is an internal convenience function sourced by `backgroundIndex()`.

Value

A list with background masking parameters. Can be one of 4 classes:

1. `bg_rect`: If `lower` and `upper` are specified.
2. `bg_sphere`: If `center` and `radius` are specified.
3. `bg_t`: If `transparent` is `TRUE` and there is an alpha channel with transparent pixels.
4. `bg_none`: If no background masking is specified (or transparency was specified but there are no transparent pixels).

Examples

```
# masking a white background:
backgroundCondition(lower = rep(0.9, 3), upper = rep(1, 3), quietly = FALSE)

# masking transparent pixels:
backgroundCondition(transparent = TRUE, alpha_channel = TRUE, quietly = FALSE)

# oops, no alpha channel:
backgroundCondition(transparent = TRUE, alpha_channel = FALSE, quietly = FALSE)

# oops, no alpha channel, but with white background as a fallback:
backgroundCondition(lower = rep(0.9, 3), upper = rep(1, 3),
                    transparent = TRUE, alpha_channel = FALSE,
                    quietly = FALSE)
```

---

backgroundIndex *Index and remove background pixels for color clustering*

Description

Largely internal function for identifying, indexing, and removing background pixels from an image.

Usage

```
backgroundIndex(img, bg_condition)
```

Arguments

`img`	An image array, preferably the output of `png::readPNG()`, `jpeg::readJPEG()`, or `link[recolorize]{readImage}`.
`bg_condition`	Background condition, output of `backgroundCondition()`.

Details

This function flattens a 3-channel image into a 2D matrix before indexing and removing background pixels to take advantage of faster indexing procedures. The `idx`, `idx_flat`, and `img_dims` elements are used to reconstruct the original and recolored images by other functions.

Value

A list with the following elements:

1. `flattened_img`: The original image, flattened into a 2D matrix (rows = pixels, columns = channels).
2. `img_dims`: Dimensions of the original image.
3. `non_bg`: Pixels from `flattened_img` that fall outside the background masking conditions. Used for further color clustering and analysis.
4. `idx`: 2D (row-column) indices for background pixels.
5. `idx_flat`: Same as `idx`, but flattened to vector order.

Examples

```
# get image path and read in image
img_path <- system.file("extdata/chongi.png", package = "recolorize")
img <- png::readPNG(img_path)
recolorize::plotImageArray(img)

# generate a white background condition
bg_condition <- backgroundCondition(lower = rep(0.9, 3),
                                    upper = rep(1, 3))

# index background pixels
bg_indexed <- backgroundIndex(img, bg_condition)

# we can reconstruct the original image from the flattened array
img2 <- bg_indexed$flattened_img
dim(img2) <- bg_indexed$img_dims

# notice the original background color (light gray) now shows
recolorize::plotImageArray(img2)
```

`blurImage` *Blur an image*

Description

Blurs an image using the one of five blur functions in `imager`. Useful for decreasing image noise.

Usage

```
blurImage(
  img,
 blur_function = c("medianblur", "isoblur", "blur_anisotropic", "boxblur", "boxblur_xy"),
  ...,
  plotting = TRUE
)
```

Arguments

`img`	An image array, as read in by png::readPNG or readImage.
`blur_function`	A string matching the name of an imager blur function. One of c("isoblur", "medianblur", "blur_anisotropic", "boxblur", "boxblur_xy").
`...`	Parameters passed to whichever `blur_function` is called.
`plotting`	Logical. Plot the blurred image next to the input for comparison?

Details

The parameters passed with the `...` argument are specific to each of the five blur functions; see their documentation for what to specify: imager::isoblur, imager::medianblur, imager::boxblur, imager::blur_anisotropic, imager::boxblur_xy. The `medianblur` and `blur_anisotropic` functions are best for preserving edges.

Value

An image array of the blurred image.

Examples

```
img_path <- system.file("extdata/fulgidissima.png", package = "recolorize")
img <- readImage(img_path)
median_img <- blurImage(img, "medianblur", n = 5, threshold = 0.5)
anisotropic_img <- blurImage(img, "blur_anisotropic",
                             amplitude = 5, sharpness = 0.1)

# save current graphical parameters:
current_par <- graphics::par(no.readonly = TRUE)
graphics::layout(matrix(1:3, nrow = 1))

plotImageArray(img, "original")
plotImageArray(median_img, "median")
plotImageArray(anisotropic_img, "anisotropic")

# and reset:
graphics::par(current_par)
```

`brick_to_array` *Convert from a RasterBrick to an array*

Description

Converts from a RasterBrick to a numeric array. Useful in going from patternize to recolorize.

Usage

```
brick_to_array(raster_brick)
```

Arguments

`raster_brick`	An object of RasterBrick class.

Details

This function is provided to convert from the RasterBrick objects provided by the alignment functions in the patternize package, e.g. `alignLan`.

Value

An image array (probably 1, 3, or 4 channels).

`cielab_coldist` *Generate a 'coldist' object for CIE Lab colors*

Description

A stopgap function for generating a pavo::coldist object from CIE Lab colors. This a pretty serious abstraction of the original intention of a `coldist` object, which is to use a combination of spectra data, visual model, and/or receptor-noise model to calculate perceived chromatic and achromatic distances between colors. Because CIE Lab color space is an approximately perceptually uniform color space for human vision, we can calculate a version of those distances for a human viewer directly from CIE Lab. A decent option if you want preliminary results, if you only care about human perception, or if you don't have access to spectral data.

Usage

```
cielab_coldist(rgbcols)
```

Arguments

`rgbcols`	An nx3 matrix of RGB colors (rows are colors and columns are R, G, and B channels).

Details

I have mixed feelings about this function and would like to replace it with something a little less hand-wavey.

Value

A pavo::coldist object with four columns: the patches being contrasted (columns 1-2), the chromatic contrast (`dS`), and the achromatic contrast (`dL`), all in units of Euclidean distance in CIE Lab space.

`cimg_to_array` *Converts from cimg to raster array*

Description

What it says it does.

Usage

```
cimg_to_array(x)
```

Arguments

`x`	A `cimg` object.

Value

A 3D array.

`classify_recolorize` *Convert a* `recolorize` *object to a* `classify` *object*

Description

Converts a recolorize object to a pavo::classify object for use in pavo.

Usage

```
classify_recolorize(recolorize_obj, imgname = "")
```

Arguments

`recolorize_obj`	A `recolorize` object.
`imgname`	Name of the image (a string).

Details

This is mostly for internal use, and hasn't been tested much.

Value

A pavo::classify object. The background patch will always be the first color (patch 1), and will be white by default.

`clean_merge_params` *Clean up parameters passed to mergeLayers*

Description

Internal function for tidiness.

Usage

```
clean_merge_params(recolorize_obj, merge_list, color_to)
```

Arguments

`recolorize_obj`	Object of `recolorize` class.
`merge_list`	List of layers to merge.
`color_to`	Argument for coloring new layers.

Value

A list of `mergeLayers` parameters in a standardized format.

`col2col` *Modified convertColor*

Description

Just like grDevices::convertColor, but with HSV as an option.

Usage

```
col2col(
  pixel_matrix,
  from = c("sRGB", "Lab", "Luv", "HSV"),
  to = c("sRGB", "Lab", "Luv", "HSV"),
  ref_white = "D65"
)
```

Arguments

`pixel_matrix`	A matrix of pixel colors, rows are pixels and columns are channels.
`from`	Color space to convert from.
`to`	Color space to convert to.
`ref_white`	Reference white.

Details

As my mother used to say: good enough for government work.

Value

A pixel matrix in the specified `to` color space.

`colorClusters` *Generate color clusters from an image*

Description

Clusters all the pixels in an image according to the specified method and returns color centers, cluster assignments, and cluster sizes.

Usage

```
colorClusters(
  bg_indexed,
  method = c("histogram", "kmeans"),
  n = 10,
  bins = 3,
  color_space = "Lab",
  ref_white = "D65",
  bin_avg = TRUE
)
```

Arguments

`bg_indexed`	A list returned by `backgroundIndex()`.
`method`	Binning scheme to use, one of either `kmeans` or `histogram`. Produce very different results (see details).
`n`	If `method = "kmeans"`, the number of colors to fit.
`bins`	If `method = "histogram"`, either the number of bins per color channel (if a single number is provided) OR a vector of length 3 with the number of bins for each channel.
`color_space`	Color space in which to cluster colors, passed to `[grDevices]{convertColor}`. One of "sRGB", "Lab", or "Luv". Default is "Lab", a perceptually uniform (for humans) color space.
`ref_white`	Reference white for converting to different color spaces. D65 (the default) corresponds to standard daylight.
`bin_avg`	Logical. Return the color centers as the average of the pixels assigned to the bin (the default), or the geometric center of the bin?

Details

`stats::kmeans()` clustering tries to find the set of `n` clusters that minimize overall distances. Histogram binning divides up color space according to set breaks; for example, bins = 2 would divide the red, green, and blue channels into 2 bins each (> 0.5 and < 0 .5), resulting in 8 possible ranges. A white pixel (RGB = 1, 1, 1) would fall into the R \> 0.5, G \> 0.5, B \> 0.5 bin. The resulting centers represent the average color of all the pixels assigned to that bin.

K-means clustering can produce more intuitive results, but because it is iterative, it will find slightly different clusters each time it is run, and their order will be arbitrary. It also tends to divide up similar colors that make up the majority of the image. Histogram binning will produce the same results every time, in the same order, and because it forces the bins to be dispersed throughout color space, tends to better pick up small color details. Bins are also comparable across images. However, this sometimes means returning empty bins (i.e. the white bin will be empty if clustering a very dark image).

Value

A list with the following elements:

1. `pixel_assignments`: A vector of color center assignments for each pixel.
2. `centers`: A matrix of color centers, in RGB color space.
3. `sizes`: The number of pixels assigned to each cluster.

Examples

```
# make a 100x100 'image' of random colors
img <- array(runif(30000), dim = c(100, 100, 3))
plotImageArray(img)

# make a background index object:
bg_indexed <- backgroundIndex(img, backgroundCondition())

# histogram clustering
hist_clusters <- colorClusters(bg_indexed, method = "hist", bins = 2)
plotColorPalette(hist_clusters$centers)

# we can use a different number of bins for each channel
uneven_clusters <- colorClusters(bg_indexed, method = "hist",
                                 bins = c(3, 2, 1))
plotColorPalette(uneven_clusters$centers)

# using kmeans
kmeans_clusters <- colorClusters(bg_indexed, method = "kmeans",
                                 n = 5)
plotColorPalette(kmeans_clusters$centers)
```

`colorClustersHist` *Cluster pixel colors using histogram binning*

Description

Clusters pixel colors by dividing color space up into specified bins, then taking the average color of all the pixels within that bin.

Usage

```
colorClustersHist(
  pixel_matrix,
  bins = 3,
  color_space = c("Lab", "sRGB", "Luv", "HSV"),
  ref_white = "D65",
  bin_avg = TRUE
)
```

Arguments

`pixel_matrix`	2D matrix of pixels to classify (rows = pixels, columns = channels).
`bins`	Number of bins for each channel OR a vector of length 3 with bins for each channel. `bins = 3` will result in 3^3 = 27 bins; `bins = c(2, 2, 3)` will result in 223 = 12 bins (2 red, 2 green, 3 blue if you're in RGB color space), etc.
`color_space`	Color space in which to cluster colors, passed to `[grDevices]{convertColor}`. One of "sRGB", "Lab", or "Luv". Default is "Lab", a perceptually uniform (for humans) color space.
`ref_white`	Reference white for converting to different color spaces. D65 (the default) corresponds to standard daylight.
`bin_avg`	Logical. Return the color centers as the average of the pixels assigned to the bin (the default), or the geometric center of the bin?

Details

Called by `colorClusters()`. See that documentation for examples.

Value

A list with the following elements:

1. `pixel_assignments`: A vector of color center assignments for each pixel.
2. `centers`: A matrix of color centers.
3. `sizes`: The number of pixels assigned to each cluster.

`colorClustersKMeans` *Cluster pixel colors using K-means clustering*

Description

Clusters pixel colors using `stats::kmeans()`.

Usage

```
colorClustersKMeans(
  pixel_matrix,
  n = 10,
  color_space = "Lab",
  ref_white = "D65"
)
```

Arguments

`pixel_matrix`	2D matrix of pixels to classify (rows = pixels, columns = channels).
`n`	Number of clusters to fit.
`color_space`	Color space in which to cluster colors, passed to `[grDevices]{convertColor}`. One of "sRGB", "Lab", "Luv", or "XYZ". Default is "Lab", a perceptually uniform (for humans) color space.
`ref_white`	Reference white for converting to different color spaces. D65 (the default) corresponds to standard daylight.

Details

Called by `colorClusters()`. See that documentation for examples.

Value

A list with the following elements:

1. `pixel_assignments`: A vector of color center assignments for each pixel.
2. `centers`: A matrix of color centers.
3. `sizes`: The number of pixels assigned to each cluster.

`colorResiduals` *Calculate squared residuals for color centers*

Description

Calculates the squared distance between each pixel and its assigned color center.

Usage

```
colorResiduals(
  pixel_matrix,
  pixel_assignments,
  centers,
  color_space = "Lab",
  metric = "euclidean",
  ref_white = "D65"
)
```

Arguments

`pixel_matrix`	2D matrix of pixels to classify (rows = pixels, columns = channels).
`pixel_assignments`	A vector of color center assignments for each pixel. Must match the order of `pixel_matrix`.
`centers`	A matrix of color centers, with rows as centers and columns as color channels. Rows are assumed to match the index values of `pixel_assignments`, e.g. a pixel assigned `1` in the assignment vector is assigned to the color in the first row of `centers`.
`color_space`	Color space in which to calculate distances. One of "sRGB", "Lab", "Luv", or "XYZ". Passed to `grDevices::convertColor()`.
`metric`	Distance metric to be used for calculating pairwise pixel distances in the given color space; passed to `stats::dist()`.
`ref_white`	Passed to `grDevices::convertColor()` if `color_space = "Lab`. Reference white for CIE Lab space.

Value

A list with the following attributes:

1. `sq_residuals`: The squared residual for every pixel in pixel_matrix.
2. `tot_residuals`: The sum of all squared residuals.
3. `avg_residual`: The average squared residual.
4. `residuals_by_center`: A list of squared residuals for every color center.
5. `avg_by_center`: The average squared residual for every color center.

Examples

```
# RGB extremes (white, black, red, green, blue, yellow, magenta, cyan)
ctrs <- matrix(c(1, 1, 1,
                 0, 0, 0,
                 1, 0, 0,
                 0, 1, 0,
                 0, 0, 1,
                 1, 1, 0,
                 1, 0, 1,
                 0, 1, 1), byrow = TRUE, ncol = 3)

# plot it
recolorize::plotColorPalette(ctrs)

# create a pixel matrix of random colors
pixel_matrix <- matrix(runif(3000), ncol = 3)

# assign pixels
# see `assignPixels` function for details
reassigned <- assignPixels(ctrs, pixel_matrix, adjust_centers = TRUE)

# find residuals from original color centers
color_residuals <- colorResiduals(pixel_matrix = pixel_matrix,
                                  pixel_assignments = reassigned$pixel_assignments,
                                  centers = ctrs)

# compare to residuals from adjusted color centers
color_residuals_adjust <- colorResiduals(pixel_matrix = pixel_matrix,
                                  pixel_assignments = reassigned$pixel_assignments,
                                  centers = reassigned$centers)
# to reset graphical parameters:
current_par <- graphics::par(no.readonly = TRUE)

layout(matrix(1:2, nrow = 2))
hist(color_residuals$sq_residuals,
 breaks = 30, border = NA, col = "tomato",
 xlim = c(0, 1), xlab = "Squared residual",
 main = "Original centers")

hist(color_residuals_adjust$sq_residuals,
breaks = 30, border = NA, col = "cornflowerblue",
xlim = c(0, 1), xlab = "Squared residual",
main = "Adjusted centers")

graphics::par(current_par)
```

`constructImage` *Generate an image from pixel assignments and color matrix*

Description

Combines a matrix of pixel assignments and a corresponding matrix of colors to make a recolored RGB image.

Usage

```
constructImage(pixel_assignments, centers, background_color = "white")
```

Arguments

`pixel_assignments`	A matrix of index values for each pixel which corresponds to `centers` (e.g. a `1` indicates that pixel is the color of the first row of `centers`). Pixels with an index value of 0 are considered background.
`centers`	An n x 3 matrix of color centers where rows are colors and columns are R, G, and B channels.
`background_color`	A numeric RGB triplet, a hex code, or a named R color for the background. Will be masked by alpha channel (and appear white in the plot window), but will be revealed if the alpha channel is removed. If the alpha channel is a background mask, this is the 'baked in' background color.

Value

An image (raster) array of the recolored image, with four channels (R, G, B, and alpha).

`editLayer` *Edit a color patch using morphological operations*

Description

Applies one of several morphological operations from `imager` to a layer of a recolorize object. Convenient for cleaning up a color patch without affecting other layers of the recolorized image. This can be used to despeckle, fill in holes, or uniformly grow or shrink a color patch.

Usage

```
editLayer(
  recolorize_obj,
  layer_idx,
  operation = "clean",
  px_size = 2,
  plotting = TRUE
)
```

Arguments

`recolorize_obj`	A recolorize object from `recolorize()`, `recluster()`, or `imposeColors()`.
`layer_idx`	A single index value (numeric) indicating which layer to edit. Corresponds to the order of the colors in the `centers` attribute of the recolorize object, and to the indices in the `pixel_assignments` attribute of the same.
`operation`	The name of an imager morphological operation to perform on the layer, passed as a string. See details.
`px_size`	The size (in pixels) of the elements to filter. If `operation` = `"shrink"` and `px_size` = 2, for example, the color patch will be shrunk by a 2-pixel radius.
`plotting`	Logical. Plot results?

Details

Current imager operations are:

- `imager::grow()`: Grow a pixset
- `imager::shrink()`: Shrink a pixset
- `imager::fill()`: Remove holes in an pixset. Accomplished by growing and then shrinking a pixset.
- `imager::clean()`: Remove small isolated elements (speckle). Accomplished by shrinking and then growing a pixset.

Value

A `recolorize` object. The `sizes`, `pixel_assignments,`, and `recolored_img` attributes will differ from the input object for the relevant color patch (layer) to reflect the edited layer.

See Also

editLayers for editing multiple layers (with multiple operations) at once; a wrapper for this function.

Examples

```
# load image and recolorize it
img <- system.file("extdata/corbetti.png", package = "recolorize")

# first do a standard color binning
init_fit <- recolorize(img, bins = 2, plotting = FALSE)

# then cluster patches by similarity
re_fit <- recluster(init_fit, cutoff = 40)

# to reset graphical parameters:
current_par <- graphics::par(no.readonly = TRUE)

# examine individual layers:
layout(matrix(1:6, nrow = 2))
layers <- splitByColor(re_fit, plot_method = "color")
```

```
# notice patch 2 (cream) - lots of stray pixels
edit_cream_layer <- editLayer(re_fit,
                              layer_idx = 2,
                              operation = "clean",
                              px_size = 3)

# shrinking and growing by the same element size gives us less flexibility, so
# we can also shrink and then grow, using different px_size arguments:
edit_green_1 <- editLayer(re_fit,
                          layer_idx = 4,
                          operation = "shrink",
                          px_size = 2)
edit_green_2 <- editLayer(edit_green_1,
                          layer_idx = 4,
                          operation = "grow",
                          px_size = 3)

# we can get pleasingly mondrian about it:
new_fit <- re_fit
for (i in 1:nrow(new_fit$centers)) {
  new_fit <- editLayer(new_fit,
                       layer_idx = i,
                       operation = "fill",
                       px_size = 5, plotting = FALSE)
}
plot(new_fit)

graphics::par(current_par)
```

`editLayers` *Edit multiple color patches using morphological operations*

Description

A wrapper for editLayer, allowing for multiple layers to be edited at once, either with the same morphological operation or specified for each layer.

Usage

```
editLayers(
  recolorize_obj,
  layer_idx = "all",
  operations = "clean",
  px_sizes = 2,
  plotting = TRUE
)
```

Arguments

`recolorize_obj`	A recolorize object from `recolorize()`, `recluster()`, or `imposeColors()`.
`layer_idx`	A numeric vector of layer indices to be edited, or `"all"` (in which case all layers are edited). Corresponds to the order of the colors in the `centers` attribute of the recolorize object, and to the indices in the `pixel_assignments` attribute of the same.
`operations`	Either a single string OR a character vector of imager morphological operation(s) to perform on the specified layer(s). If this is shorter than `layer_idx`, it is repeated to match the length of `layer_idx`.
`px_sizes`	The size(s) (in pixels) of the elements to filter. Either a single number OR a numeric vector. If shorter than `layer_idx`, it is repeated to match the length of `layer_idx`. If `operation = "shrink"` and `px_size = 2`, for example, the color patch will be shrunk by a 2-pixel radius.
`plotting`	Logical. Plot results?

Details

Current imager operations are:

- `imager::grow()`: Grow a pixset
- `imager::shrink()`: Shrink a pixset
- `imager::fill()`: Remove holes in an pixset. Accomplished by growing and then shrinking a pixset.
- `imager::clean()`: Remove small isolated elements (speckle). Accomplished by shrinking and then growing a pixset.

Value

A `recolorize` object. The `sizes`, `pixel_assignments`, , and `recolored_img` attributes will differ from the input object for the relevant color patches (layers) to reflect their changes.

See Also

editLayer for editing a single layer at a time.

Examples

```
# load image and recolorize it
img <- system.file("extdata/corbetti.png", package = "recolorize")

# first do a standard color binning
init_fit <- recolorize(img, bins = 2, plotting = FALSE)

# then cluster patches by similarity
re_fit <- recluster(init_fit, cutoff = 40)

# to reset graphical parameters:
current_par <- graphics::par(no.readonly = TRUE)
```

```
# examine individual layers:
layout(matrix(1:6, nrow = 2))
layers <- splitByColor(re_fit, plot_method = "color")

# we can clean them all using the same parameters...
edited_fit <- editLayers(re_fit, layer_idx = "all",
                         operations = "clean",
                         px_sizes = 2, plotting = TRUE)
# ...but some of those patches don't look so good

# we can use different px_sizes for each layer:
edited_fit_2 <- editLayers(re_fit, layer_idx = "all",
                           operations = "clean",
                           px_sizes = c(1, 3, 1,
                                        2, 1, 2),
                           plotting = TRUE)

# better yet, we can fill some layers and clean others:
edited_fit_3 <- editLayers(re_fit, layer_idx = "all",
                           operations = c("fill", "clean",
                                          "fill", "fill",
                                          "fill", "clean"),
                           px_sizes = c(2, 3,
                                        2, 2,
                                        4, 2))

# or you could just get weird:
edited_fit_3 <- editLayers(re_fit, layer_idx = c(1:6),
                           operations = c("fill", "clean"),
                           px_sizes = c(10, 20))

# reset graphical parameters:
graphics::par(current_par)
```

`expand_recolorize` *Expand aspects of a recolorize object for other functions*

Description

Expand aspects of a recolorize object for other functions

Usage

```
expand_recolorize(
  recolorize_obj,
  original_img = FALSE,
  recolored_img = FALSE,
  sizes = FALSE
)
```

Arguments

`recolorize_obj`	A `recolorize` object.
`original_img`	Logical. Return original image as numeric array?
`recolored_img`	Logical. Return recolored image as numeric array?
`sizes`	Logical. Return cluster sizes (as number of pixels)?

Value

A `recolorize` object with the indicated additional elements, as well as the original elements.

`hclust_color` *Plot and group colors by similarity*

Description

A wrapper for stats::hclust for clustering colors by similarity. This works by converting a matrix of RGB centers to a given color space (CIE Lab is the default), generating a distance matrix for those colors in that color space (or a subset of channels of that color space), clustering them, and plotting them with labels and colors. If either a cutoff or a final number of colors is provided and `return_list = TRUE`, function also returns a list of which color centers to combine.

Usage

```
hclust_color(
  rgb_centers,
  dist_method = "euclidean",
  hclust_method = "complete",
  channels = 1:3,
  color_space = "Lab",
  ref_white = "D65",
  cutoff = NULL,
  n_final = NULL,
  return_list = TRUE,
  plotting = TRUE
)
```

Arguments

`rgb_centers`	A matrix of RGB centers. Rows are centers and columns are R, G, and B values.
`dist_method`	Method passed to stats::dist. One of "euclidean", "maximum", "manhattan", "canberra", "binary" or "minkowski".
`hclust_method`	Method passed to stats::hclust. One of "ward.D", "ward.D2", "single", "complete", "average" (= UPGMA), "mcquitty" (= WPGMA), "median" (= WPGMC) or "centroid" (= UPGMC).

`channels`	Numeric: which color channels to use for clustering. Probably some combination of 1, 2, and 3, e.g., to consider only luminance and blue-yellow (b-channel) distance in CIE Lab space, `channels = c(1, 3` (L and b).
`color_space`	Color space in which to do the clustering.
`ref_white`	Reference white for converting to different color spaces. D65 (the default) corresponds to standard daylight. See grDevices::convertColor.
`cutoff`	Either `NULL` or a numeric cutoff passed to stats::cutree. Distance below which to combine clusters, i.e. height at which the tree should be cut.
`n_final`	Numeric. Desired number of groups. Overrides `cutoff` if both are provided.
`return_list`	Logical. Return a list of new group assignments from the `cutoff` or `n_final` values?
`plotting`	Logical. Plot a colored dendrogram?

Details

This is mostly useful in deciding where and in which color space to place a cutoff for a `recolorize` object, since it is very fast. It is called by recluster when combining layers by similarity.

Value

A list of group assignments (i.e. which centers belong to which groups), if `return_list = TRUE`.

See Also

recluster

Examples

```
# 50 random RGB colors
rgb_random <- matrix(runif(150), nrow = 50, ncol = 3)

# default clustering (Lab space):
hclust_color(rgb_random, return_list = FALSE)

# clustering in RGB space (note change in Y-axis scale):
hclust_color(rgb_random, color_space = "sRGB", return_list = FALSE)

# clustering using only luminance:
hclust_color(rgb_random, channels = 1, return_list = FALSE)

# or only red-green ('a' channel):
hclust_color(rgb_random, channels = 2, return_list = FALSE)

# or only blue-yellow ('b' channel(:
hclust_color(rgb_random, channels = 3, return_list = FALSE)

# use a cutoff to get groups:
groups <- hclust_color(rgb_random, cutoff = 100)
print(groups)
```

`imDist` *Calculates the distance between non-transparent pixels in images*

Description

Compares two versions of the same image (probably original and recolored) by calculating the color distance between the colors of each pair of pixels.

Usage

```
imDist(
  im1,
  im2,
  color_space = c("Lab", "sRGB", "XYZ", "Luv"),
  ref_white = "D65",
  metric = "euclidean",
  plotting = TRUE,
  palette = "default",
  main = "",
  ...
)
```

Arguments

`im1, im2`	Images to compare; must have the same dimensions. Distances will be calculated between each pair of non-transparent pixels.
`color_space`	Color space in which to calculate distances. One of "sRGB", "Lab", "Luv", or "XYZ". Passed to `grDevices::convertColor()`.
`ref_white`	Passed to `grDevices::convertColor()` if `color_space = "Lab`. Reference white for CIE Lab space.
`metric`	Distance metric to be used for calculating pairwise pixel distances in the given color space; passed to `stats::dist()`.
`plotting`	Logical. Plot heatmap of color distances?
`palette`	If plotting, the color palette to be used. Default is blue to red (`colorRamps::blue2red(100)`).
`main`	Plot title.
`...`	Parameters passed to `graphics::image()`.

Value

A matrix of the same dimensions as the original images, with the distance between non-transparent pixels at each pixel coordinate. Transparent pixels are returned as `NA`.

Examples

```
fulgidissima <- system.file("extdata/fulgidissima.png",
                             package = "recolorize")
fulgidissima <- png::readPNG(fulgidissima)
# make an initial histogram fit
# this doesn't look great:
fulgidissima_2bin <- recolorize(fulgidissima, "hist", bins = 2)

# we can compare with the original image by creating the recolored
# image from the colormap
recolored_2bin <- constructImage(fulgidissima_2bin$pixel_assignments,
                                fulgidissima_2bin$centers)
dist_2bin <- imDist(im1 = fulgidissima,
                    im2 = recolored_2bin)

# using 3 bins/channel looks much better:
fulgidissima_3bin <- recolorize(fulgidissima, "hist", bins = 3)

# and we can see that on the heatmap:
recolored_3bin <- constructImage(fulgidissima_3bin$pixel_assignments,
                                fulgidissima_3bin$centers)
dist_3bin <- imDist(im1 = fulgidissima,
                    im2 = recolored_3bin)

# default behavior is to set the color range to the range of distances
# in a single matrix; to compare two different fits, we have to provide
# the same `zlim` scale for both
r <- range(c(dist_2bin, dist_3bin), na.rm = TRUE)

# to reset graphical parameters:
current_par <- graphics::par(no.readonly = TRUE)

# now we can plot them to compare the fits:
layout(matrix(1:2, nrow = 1))
imHeatmap(dist_2bin, range = r)
imHeatmap(dist_3bin, range = r)

# we can also use other color spaces:
rgb_3bin <- imDist(fulgidissima,
                   recolored_3bin,
                   color_space = "sRGB")

# looks oddly worse, but to keep things in perspective,
# you can set the range to the maximum color distance in RGB space:
imHeatmap(rgb_3bin, range = c(0, sqrt(3)))
# not useful for troubleshooting, but broadly reassuring!

# reset:
graphics::par(current_par)
```

---

imHeatmap *Plot a heatmap of a matrix of color distances*

Description

Plots the output of `imDist()` as a heatmap.

Usage

```
imHeatmap(
  mat,
  palette = "default",
  main = "",
  range = NULL,
  legend = TRUE,
  ...
)
```

Arguments

`mat`	A color distance matrix, preferably output of `imDist()`.
`palette`	The color palette to be used. Default is blue to red (`colorRamps::blue2red(100)`).
`main`	Plot title.
`range`	Range for heatmap values. Defaults to the range of values in the matrix, but should be set to the same range for all images if comparing heatmaps.
`legend`	Logical. Add a continuous color legend?
`...`	Parameters passed to `graphics::image()`.

Value

Nothing; plots a heatmap of the color residuals.

Examples

```
chongi <- system.file("extdata/chongi.png", package = "recolorize")
chongi <- png::readPNG(chongi)
chongi_k <- recolorize(chongi, "k", n = 5)

recolored_chongi <- constructImage(chongi_k$pixel_assignments,
                                   chongi_k$centers)
d <- imDist(chongi,
            recolored_chongi, plotting = FALSE)

# original flavor
imHeatmap(d)

# bit offputting
```

```
imHeatmap(d, palette = colorRamps::ygobb(100))

# just dreadful
imHeatmap(d, palette = colorRamps::primary.colors(100))
```

`imposeColors` *Recolor an image to a provided set of colors*

Description

Takes an image and a set of color centers, and assigns each pixel to the most similar provided color. Useful for producing a set of images with identical colors.

Usage

```
imposeColors(
  img,
  centers,
  adjust_centers = TRUE,
  color_space = "sRGB",
  ref_white = "D65",
  lower = NULL,
  upper = NULL,
  transparent = TRUE,
  resid = FALSE,
  resize = NULL,
  rotate = NULL,
  plotting = TRUE,
  horiz = TRUE,
  cex_text = 1.5,
  scale_palette = TRUE
)
```

Arguments

`img`	Path to the image (a character vector) or a 3D image array as read in by `png::readPNG()` `{readImage}`.
`centers`	Colors to map to, as an n x 3 matrix (rows = colors, columns = channels).
`adjust_centers`	Logical. After pixel assignment, should the returned colors be the average color of the pixels assigned to that cluster, or the original colors?
`color_space`	Color space in which to minimize distances. One of "sRGB", "Lab", "Luv", "HSV", or "XYZ". Default is "Lab", a perceptually uniform (for humans) color space.
`ref_white`	Reference white for converting to different color spaces. D65 (the default) corresponds to standard daylight.
`lower, upper`	RGB triplet ranges for setting a bounding box of pixels to mask. See details.

`transparent`	Logical. Treat transparent pixels as background? Requires an alpha channel (PNG).
`resid`	Logical. Return a list of different residual metrics to describe the goodness of fit?
`resize`	A value between 0 and 1 for resizing the image (ex. `resize = 0.5` will reduce image size by 50%). Recommended for large images as it can speed up analysis considerably. See details.
`rotate`	Degrees to rotate the image clockwise.
`plotting`	Logical. Plot recolored image & color palette?
`horiz`	Logical for plotting. Plot output image and color palette side by side (`TRUE`) or stacked vertically (`FALSE`)?
`cex_text`	If `plotting = TRUE` and `scale_palette = FALSE`, size of text to display on the color palette numbers.
`scale_palette`	Logical. If plotting, plot colors in the color palette proportional to the size of each cluster?

Details

Background masking: `lower`, `upper`, and `transparent` are all background masking conditions. Transparency is unambiguous and so tends to produce cleaner results, but the `lower` and `upper` bounds can be used instead to treat pixels in a specific color range as the background. For example, to ignore white pixels (RGB = 1, 1, 1), you might want to mask all pixels whose R, G, and B values exceed 0.9. In that case, `lower = c(0.9, 0.9, 0.9)` and `upper = c(1, 1, 1)`. Regardless of input background, recolored images are returned with transparent backgrounds by adding an alpha channel if one does not already exist.

Resizing: The speed benefits of downsizing images are fairly obvious (fewer pixels = fewer operations). Because recoloring the images simplifies their detail anyways, downsizing prior to recoloring doesn't run a very high risk of losing important information. A general guideline for resizing is that any distinguishable features of interest should still take up at least 2 pixels (preferably with a margin of error) in the resized image.

Value

A list with the following attributes:

1. `original_img`: The original image, as a raster.
2. `centers`: A matrix of color centers. If `adjust_centers = FALSE`, this will be identical to the input `centers`.
3. `sizes`: The number of pixels assigned to each color cluster.
4. `pixel_assignments`: A vector of color center assignments for each pixel.
5. `call`: The call(s) used to generate the `recolorize` object.

Examples

```
# RGB extremes (white, black, red, green, blue, yellow, magenta, cyan)
ctrs <- matrix(c(1, 1, 1,
                 0, 0, 0,
                 1, 0, 0,
                 0, 1, 0,
                 0, 0, 1,
                 1, 1, 0,
                 1, 0, 1,
                 0, 1, 1), byrow = TRUE, ncol = 3)

# plot it
recolorize::plotColorPalette(ctrs)

# get image paths
ocellata <- system.file("extdata/ocellata.png", package = "recolorize")

# map to rgb extremes
ocellata_fixed <- recolorize::imposeColors(ocellata, ctrs,
                                            adjust_centers = FALSE)

# looks much better if we recalculate the centers from the image
ocellata_adjusted <- recolorize::imposeColors(ocellata, ctrs,
                                           adjust_centers = TRUE)

# we can map one image to extracted colors from another image
# extract ocellata colors
ocellata_colors <- recolorize(ocellata)

# map fulgidissima to ocellata colors
fulgidissima <- system.file("extdata/fulgidissima.png",
                             package = "recolorize")

fulgidissma_ocellata <- recolorize::imposeColors(fulgidissima,
                       ocellata_colors$centers,
                       adjust_centers = FALSE)
```

`labelCol` *Change colors of dendrogram tips*

Description

Internal function for recluster plotting.

Usage

```
labelCol(x, hex_cols, pch = 20, cex = 2)
```

Arguments

`x`	Leaf of a dendrogram.
`hex_cols`	Hex color codes for colors to change to.
`pch`	The type of point to draw.
`cex`	The size of the point.

Value

An `hclust` object with colored tips.

`match_colors` *Reorder a color palette to best match a reference palette*

Description

Often for batch processing purposes, it is important to ensure that color centers fit using different methods are in the same order. This function reorders a provided color palette (`match_palette`) according a provided reference palette (`reference_palette`) by minimizing their overall distance using the Hungarian algorithm as implemented by clue::solve_LSAP.

Usage

```
match_colors(reference_palette, match_palette, plotting = FALSE)
```

Arguments

`reference_palette`	The palette whose order to match. Either a character vector of colors (hex codes or color names) or an nx3 matrix in **sRGB color space**.
`match_palette`	The palette to reorder, same formats as `reference_palette`
`plotting`	Logical. Plot the ordered palettes?

Details

If the color palettes are wildly different, the returned order may not be especially meaningful.

Value

A vector of color orders for `match_palette`.

See Also

reorder_colors

Examples

```
ref_palette <- c("mediumblue", "olivedrab", "tomato2", "beige", "chocolate4")
match_palette <- c("#362C34", "#E4D3A9", "#AA4E47", "#809C35", "#49468E")
match_colors(ref_palette, match_palette, plotting = TRUE)
```

`medianColors` *Change color centers to median color of all pixels assigned to it*

Description

By default, recolorize sets the centers of each color patch to the average (mean) color of all pixels assigned to it. This can sometimes result in colors that look washed out, especially in cases where a region is very shiny (e.g. black with white reflective highlights will average to grey). In these cases, switching to median colors may be either more accurate or more visually pleasing.

Usage

```
medianColors(recolorize_obj, plotting = TRUE)
```

Arguments

`recolorize_obj`	A `recolorize` class object.
`plotting`	Logical. Plot results?

Value

A `recolorize` object, with median colors instead of average colors in the `centers` attribute.

`mergeLayers` *Merge layers in a recolorized image*

Description

Merges specified layers in a recolorized image. This is a good option if you want to manually specify which layers to merge (and what color to make the resulting merged layer); it's also called on by other `recolorize` functions like `recluster()` to merge layers that have been identified as highly similar in color using a given distance metric.

Usage

```
mergeLayers(
  recolorize_obj,
  merge_list = NULL,
  color_to = "weighted average",
  plotting = TRUE,
  remove_empty_centers = FALSE
)
```

Arguments

`recolorize_obj`	An object of class "recolorize", such as from `recolorize()`, `recluster()`, or `imposeColors()`.
`merge_list`	A list of numeric vectors specifying which layers to merge. Layers not included in this list are unchanged. See examples.
`color_to`	Color(s) for the merged layers. See examples.
`plotting`	Logical. Plot the results of the layer merging next to the original color fit for comparison?
`remove_empty_centers`	Logical. Remove empty centers with size = 0? Retaining empty color centers can be helpful when batch processing.

Details

Colors can be supplied as numeric RGB triplets (e.g. `c(1, 1, 1)` for white), a valid R color name (`"white"`), or a hex code (`"#FFFFFF`). Alternatively, `color_to = "weighted average"` will set the merged layer to the average color of the layers being merged, weighted by their relative size. Must be either a single value or a vector the same length as `merge_list`. If a single color is supplied, then all merged layers will be set to that color (so this really is only useful if you're already merging those layers into a single layer).

Value

A `recolorize` class object with merged layers. The order of the returned layers depends on `merge_list`: the first layers will be any not included in the list, followed by the new merged layers. If you start with layers 1-8 and merge layers 4 & 5 and 7 & 8, the returned 5 layers will be, in order and in terms of the original layers: 1, 2, 3, 6, 4 & 5 (merged), 7 & 8 (merged). This is probably easiest to see in the examples.

Examples

```
# image path:
img <- system.file("extdata/corbetti.png", package = "recolorize")

# initial fit, 8 bins:
init_fit <- recolorize(img)
# redundant green, red, and blue clusters

# to make it easier to see, we can plot the numbered palette:
plot(init_fit)

# based on visual inspection, we should merge:
mlist <- list(c(3, 5),
              c(4, 7),
              c(6, 8))

# we can merge with that list, leaving layers 1 & 2 intact:
vis_merge <- mergeLayers(init_fit,
                         merge_list = mlist)
```

```
# we can include layers 1 & 2 as their own list elements,
# leaving them intact (result is identical to above):
mlist2 <- list(1, 2,
               c(3, 5),
               c(4, 7),
               c(6, 8))
redundant_merge <- mergeLayers(init_fit,
                               merge_list = mlist2)

# we can also swap layer order this way without actually merging layers:
swap_list <- list(2, 5, 3, 4, 1)
swap_layers <- mergeLayers(redundant_merge,
                           merge_list = swap_list)

# merging everything but the first layer into a single layer,
# and making that merged layer orange (result looks
# a bit like a milkweed bug):
milkweed_impostor <- mergeLayers(init_fit,
                                 merge_list = list(c(2:8)),
                                 color_to = "orange")

# we can also shuffle all the layer colors while
# leaving their geometry intact:
centers <- vis_merge$centers
centers <- centers[sample(1:nrow(centers), nrow(centers)), ]
shuffle_layers <- mergeLayers(vis_merge,
                              merge_list = as.list(1:5),
                              color_to = centers)
# (this is not really the intended purpose of this function)
```

`pixelAssignMatrix` *Make pixel assignment matrix for recoloring*

Description

Internal function. Generates a sort of 'paint-by-numbers' matrix, where each cell is the index of the color in the color centers matrix to which that pixel is assigned. An index of 0 indicates a background pixel.

Usage

```
pixelAssignMatrix(bg_indexed, color_clusters)
```

Arguments

`bg_indexed`	An object returned by `backgroundIndex()`.
`color_clusters`	An object returned by `colorClusters()`.

Value

A matrix of pixel color assignments (`pixel_assignments`) and a corresponding dataframe of color centers (`centers`).

`plot.recolorize` *Plot recolorized image results*

Description

S3 plotting method for objects of class `recolorize`. Plots a side-by-side comparison of an original image and its recolorized version, plus the color palette used for recoloring.

Usage

```
## S3 method for class 'recolorize'
plot(x, ..., plot_original = TRUE, horiz = TRUE, cex_text = 2, sizes = FALSE)
```

Arguments

`x`	An object of class `recolorize`, such as returned by `recolorize()`, `recluster()`, `imposeColors()`, etc.
`...`	further arguments passed to `plot`.
`plot_original`	Logical. Plot the original image for comparison?
`horiz`	Logical. Should plots be stacked vertically or horizontally?
`cex_text`	Text size for printing color indices. Plotting parameters passed to `[recolorize]{plotColorPalette}`.
`sizes`	Logical. If `TRUE`, color palette is plotted proportional to the size of each color. If `FALSE`, all colors take up an equal amount of space, and their indices are printed for reference.

Value

No return value; plots the original image, recolored image, and color palette.

Examples

```
corbetti <- system.file("extdata/corbetti.png",
                         package = "recolorize")

corbetti_recolor <- recolorize(corbetti, method = "hist",
                                         bins = 2, plotting = FALSE)

# unscaled color palette
plot(corbetti_recolor)

# scaled color palette
plot(corbetti_recolor, sizes = TRUE)
```

`plot.recolorizeVector` *Plot a* `recolorizeVector` *object*

Description

Plots an object generated by recolorizeVector.

Usage

```
## S3 method for class 'recolorizeVector'
plot(x, ...)
```

Arguments

`x`	Object returned by recolorizeVector.
`...`	Further arguments passed to graphics::plot.

Value

No return value; plots `recolorizeVector` as polygons.

`plotColorClusters` *Plot color clusters in a color space*

Description

Plots color clusters in a 3D color space.

Usage

```
plotColorClusters(
  centers,
  sizes,
  scaling = 10,
  plus = 0,
  color_space = "sRGB",
  phi = 35,
  theta = 60,
  alpha = 0.5,
  ...
)
```

Arguments

`centers`	A matrix of color centers, with rows for centers and columns as channels. These are interpreted as coordinates.
`sizes`	A vector of color sizes. Can be relative or absolute; it's going to be scaled for plotting.
`scaling`	Factor for scaling the cluster sizes. If your clusters are way too big or small on the plot, tinker with this.
`plus`	Value to add to each scaled cluster size; can be helpful for seeing small or empty bins when they are swamped by larger clusters.
`color_space`	The color space of the centers. Important for setting the axis ranges and for converting the colors into hex codes for plotting. The function assumes that the `centers` argument is already in this color space.
`phi, theta`	Viewing angles (in degrees).
`alpha`	Transparency (0-1 range).
`...`	Further parameters passed to plot3D::scatter3D.

Details

This function does very little on your behalf (e.g. labeling the axes, setting the axis ranges, trying to find nice scaling parameters, etc). You can pass those parameters using the ... function to plot3D::scatter3D, which is probably a good idea.

Value

Nothing; plots a 3D scatterplot of color clusters, with corresponding colors and sizes.

Examples

```
corbetti <- system.file("extdata/corbetti.png", package = "recolorize")
init_fit <- recolorize(corbetti,
                       color_space = "Lab",
                       method = "k",
                       n = 30)

# we still have to convert to Lab color space first, since the centers are always RGB:
centers <- grDevices::convertColor(init_fit$centers, "sRGB", "Lab")
plotColorClusters(centers, init_fit$sizes,
                  scaling = 25,
                  color_space = "Lab",
                  xlab = "Luminance",
                  ylab = "a (red-green)",
                  zlab = "b (blue-yellow)",
                  cex.lab = 0.5)
```

`plotColorPalette` *Plot a color palette*

Description

Plots a color palette as a single bar, optionally scaling each color to a vector of sizes.

Usage

```
plotColorPalette(centers, sizes = NULL, cex_text = 2, horiz = TRUE, ...)
```

Arguments

`centers`	Colors to plot in palette. Accepts either a character vector of hex codes or an n x 3 matrix (rows = colors, columns = channels). Assumes RGB in 0-1 range.
`sizes`	An optional numeric vector of sizes for scaling each color. If no sizes are provided, colors are plotted in equal proportions.
`cex_text`	Size of the numbers displayed on each color, relative to the default. Passed to `graphics::barplot()`. Text is only plotted if `sizes` = `NULL`. `cex_text` = `0` will remove numbering.
`horiz`	Logical. Should the palette be plotted vertically or horizontally?
`...`	Additional parameters passed to `graphics::barplot()`.

Details

`plotColorPalette` does not reorder or convert colors between color spaces, so users working in other colorspaces should convert to RGB before plotting.

Value

No return value; plots a rectangular color palette.

Examples

```
# plot 10 random colors
rand_colors <- matrix(runif(30), ncol = 3)
plotColorPalette(rand_colors)

# plot 10 random colors with arbitrary sizes
sizes <- runif(10, max = 1000)
plotColorPalette(rand_colors, sizes = sizes)

# reorder to plot smallest to largest
size_order <- order(sizes)
plotColorPalette(rand_colors[size_order, ],
                 sizes[size_order])
```

```
# plot a vector of hex colors, turn off numbering
hex_colors <- rgb(rand_colors)
plotColorPalette(hex_colors, cex_text = 0)
```

`plotImageArray` *Plot a 3D array as an RGB image*

Description

Does what it says on the tin. An extremely simple wrapper for `graphics::rasterImage()`, but maintains aspect ratio, removes axes, and reduces margins for cleaner plotting.

Usage

```
plotImageArray(rgb_array, main = "", ...)
```

Arguments

`rgb_array`	A 3D array of RGB values. Preferably output from `png::readPNG()`, `jpeg::readJPEG()`, recoloredImage, constructImage, or raster_to_array.
`main`	Optional title for plot.
`...`	Parameters passed to graphics::plot.

Value

No return value; plots image.

Examples

```
# make a 100x100 image of random colors
random_colors <- array(runif(100 * 100 * 3),
                       dim = c(100, 100, 3))
recolorize::plotImageArray(random_colors)

# we can also plot...a real image
corbetti <- system.file("extdata/corbetti.png",
                       package = "recolorize")
img <- png::readPNG(corbetti)
plotImageArray(img)
```

`raster_to_array` *Convert from a (small-r) raster object to an RGB array*

Description

Recreates the original numeric array from a `raster` object created by grDevices::as.raster. Not to be confused with the `Raster*` classes used by the `raster` package.

Usage

```
raster_to_array(raster_obj, alpha = TRUE)
```

Arguments

`raster_obj`	A matrix of hex codes as output by grDevices::as.raster.
`alpha`	Logical. If there is an alpha channel, retain it in the array?

Value

A numeric RGB array (0-1 range).

`readImage` *Read in an image as a 3D array*

Description

Reads in and processes an image as a 3D array. Extremely simple wrapper for `imager::load.image()`, but it strips the depth channel (resulting in a 3D, not 4D, array). This will probably change.

Usage

```
readImage(img_path, resize = NULL, rotate = NULL)
```

Arguments

`img_path`	Path to the image (a string).
`resize`	Fraction by which to reduce image size. Important for speed.
`rotate`	Number of degrees to rotate the image.

Value

A 3D RGB array (pixel rows x pixel columns x color channels). RGB channels are all scaled 0-1, not 0-255.

Examples

```
corbetti <- system.file("extdata/corbetti.png", package = "recolorize")
img <- readImage(corbetti)
plotImageArray(img)
```

`recluster` *Recluster color centers based on color similarity*

Description

Color mapping (as with k-means or binning) often requires over-clustering in order to recover details in an image. This can result in larger areas of relatively uniform color being split into multiple colors, or in regions with greater variation (due to lighting, shape, reflection, etc) being split into multiple colors. This function clusters the color centers by visual similarity (in CIE Lab space), then returns the re-clustered object. Users can either set a similarity cutoff or a final number of colors. See examples.

Usage

```
recluster(
  recolorize_obj,
  dist_method = "euclidean",
  hclust_method = "complete",
  channels = 1:3,
  color_space = "Lab",
  ref_white = "D65",
  cutoff = 60,
  n_final = NULL,
  plot_hclust = TRUE,
  refit_method = c("imposeColors", "mergeLayers"),
  resid = FALSE,
  plot_final = TRUE,
  color_space_fit = "sRGB"
)
```

Arguments

`recolorize_obj`	A recolorize object from `recolorize()`, `recluster()`, or `imposeColors()`.
`dist_method`	Method passed to stats::dist for calculating distances between colors. One of "euclidean", "maximum", "manhattan", "canberra", "binary" or "minkowski".
`hclust_method`	Method passed to stats::hclust for clustering colors by similarity. One of "ward.D", "ward.D2", "single", "complete", "average" (= UPGMA), "mcquitty" (= WPGMA), "median" (= WPGMC) or "centroid" (= UPGMC).
`channels`	Numeric: which color channels to use for clustering. Probably some combination of 1, 2, and 3, e.g., to consider only luminance and blue-yellow (b-channel) distance in CIE Lab space, channels = c(1, 3) (L and b).

`color_space`	Color space in which to cluster centers, passed to `[grDevices]{convertColor}`. One of "sRGB", "Lab", or "Luv". Default is "Lab", a perceptually uniform (for humans) color space.
`ref_white`	Reference white for converting to different color spaces. D65 (the default) corresponds to standard daylight.
`cutoff`	Numeric similarity cutoff for grouping color centers together. The range and value will depend on the chosen color space (see below), but the default is in absolute Euclidean distance in CIE Lab space, which means it is greater than 0-100, but cutoff values between 20 and 80 will usually work best. See details.
`n_final`	Final number of desired colors; alternative to specifying a similarity cutoff. Overrides `cutoff` if provided.
`plot_hclust`	Logical. Plot the hierarchical clustering tree for color similarity? Helpful for troubleshooting a cutoff.
`refit_method`	Method for refitting the image with the new color centers. One of either "imposeColors" or "mergeLayers". `imposeColors()` refits the original image using the new colors (slow but often better results). `mergeLayers()` merges the layers of the existing recolored image. This is faster since it doesn't require a new fit, but can produce messier results.
`resid`	Logical. Get final color fit residuals with `colorResiduals()`?
`plot_final`	Logical. Plot the final color fit?
`color_space_fit`	Passed to `imposeColors()`. What color space should the image be reclustered in?

Details

This function is fairly straightforward: the RGB color centers of the recolorize object are converted to CIE Lab color space (which is approximately perceptually uniform for human vision), clustered using `stats::hclust()`, then grouped using `stats::cutree()`. The resulting groups are then passed as the assigned color centers to `imposeColors()`, which re-fits the *original* image using the new centers.

The similarity cutoff does not require the user to specify the final number of colors, unlike k-means or `n_final`, meaning that the same cutoff could be used for multiple images (with different numbers of colors) and produce relatively good fits. Because the cutoff is in absolute Euclidean distance in CIE Lab space for sRGB colors, the possible range of distances (and therefore cutoffs) is from 0 to >200. The higher the cutoff, the more dissimilar colors will be grouped together. There is no universally recommended cutoff; the same degree of color variation due to lighting in one image might be biologically relevant in another.

Value

A `recolorize` object with the re-fit color centers.

Examples

```
# get an image
corbetti <- system.file("extdata/corbetti.png", package = "recolorize")
```

```
# too many color centers
recolored_corbetti <- recolorize(corbetti, bins = 2)

# just enough!
# check previous plot for clustering cutoff
recluster_obj <- recluster(recolored_corbetti,
                           cutoff = 45,
                           plot_hclust = TRUE,
                           refit_method = "impose")

# we get the same result by specifying n_final = 5
recluster_obj <- recluster(recolored_corbetti,
                           n_final = 5,
                           plot_hclust = TRUE)
```

`recoloredImage` *Get recolored image from a recolorize object*

Description

`recolorize` objects use a numeric color map and a matrix of color centers to make recolored images, since this is a lighter weight and more flexible format. This function generates a colored image from those values for plotting.

Usage

```
recoloredImage(recolorize_obj, type = c("array", "raster"))
```

Arguments

`recolorize_obj`	An object of class `recolorize`. Must include a pixel assignment matrix and matrix of color centers.
`type`	Type of image to return. One of either "array" or "raster". Arrays are numeric RGB arrays (larger, but easier to do operations on), rasters are matrices of hex codes (smaller, only really good for plotting).

Value

A numeric image array (if `type = array`) or a matrix of hex codes (if `type = raster`).

`recolorize` *Simplify the colors of an image*

Description

Clusters the colors in an RGB image according to a specified method, then recolors that image to the simplified color scheme.

Usage

```
recolorize(
  img,
  method = c("histogram", "kmeans"),
  bins = 2,
  n = 5,
  color_space = "sRGB",
  ref_white = "D65",
  lower = NULL,
  upper = NULL,
  transparent = TRUE,
  resid = FALSE,
  resize = NULL,
  rotate = NULL,
  plotting = TRUE,
  horiz = TRUE,
  cex_text = 1.5,
  scale_palette = TRUE,
  bin_avg = TRUE
)
```

Arguments

`img`	Path to the image (a character vector) or a 3D image array as read in by `png::readPNG()` `{readImage}`.
`method`	Method for clustering image colors. One of either `histogram` or `kmeans`. See details.
`bins`	If `method = "histogram"`, either the number of bins per color channel (if a single number is provided) OR a vector of length 3 with the number of bins for each channel.
`n`	If `method = "kmeans"`, the number of color clusters to fit.
`color_space`	Color space in which to minimize distances, passed to `[grDevices]{convertColor}`. One of "sRGB", "Lab", or "Luv". Default is "Lab", a perceptually uniform (for humans) color space.
`ref_white`	Reference white for converting to different color spaces. D65 (the default) corresponds to standard daylight.

`lower`, `upper`	RGB triplet ranges for setting a bounding box of pixels to mask. See details.
`transparent`	Logical. Treat transparent pixels as background? Requires an alpha channel (PNG).
`resid`	Logical. Return a list of different residual metrics to describe the goodness of fit?
`resize`	A value between 0 and 1 for resizing the image (ex. `resize = 0.5` will reduce image size by 50%). Recommended for large images as it can speed up analysis considerably. See details.
`rotate`	Degrees to rotate the image clockwise.
`plotting`	Logical. Plot recolored image & color palette?
`horiz`	Logical for plotting. Plot output image and color palette side by side (`TRUE`) or stacked vertically (`FALSE`)?
`cex_text`	If `plotting = TRUE` and `scale_palette = FALSE`, size of text to display on the color palette numbers.
`scale_palette`	Logical. If plotting, plot colors in the color palette proportional to the size of each cluster?
`bin_avg`	Logical. Return the color centers as the average of the pixels assigned to the bin (the default), or the geometric center of the bin?

Details

Method for color clustering: `stats::kmeans()` clustering tries to find the set of `n` clusters that minimize overall distances. Histogram binning divides up color space according to set breaks; for example, bins = 2 would divide the red, green, and blue channels into 2 bins each (> 0.5 and < 0 .5), resulting in 8 possible ranges. A white pixel (RGB = 1, 1, 1) would fall into the R > 0.5, G > 0.5, B > 0.5 bin. The resulting centers represent the average color of all the pixels assigned to that bin.

K-means clustering can produce more intuitive results, but because it is iterative, it will find slightly different clusters each time it is run, and their order will be arbitrary. It also tends to divide up similar colors that make up the majority of the image. Histogram binning will produce the same results every time, in the same order, and because it forces the bins to be dispersed throughout color space, tends to better pick up small color details. Bins are also comparable across images. However, this sometimes means returning empty bins (i.e. the white bin will be empty if clustering a very dark image).

Background masking: `lower`, `upper`, and `transparent` are all background masking conditions. Transparency is unambiguous and so tends to produce cleaner results, but the `lower` and `upper` bounds can be used instead to treat pixels in a specific color range as the background. For example, to ignore white pixels (RGB = 1, 1, 1), you might want to mask all pixels whose R, G, and B values exceed 0.9. In that case, `lower = c(0.9, 0.9, 0.9)` and `upper = c(1, 1, 1)`. Regardless of input background, recolored images are returned with transparent backgrounds by adding an alpha channel if one does not already exist.

Resizing: The speed benefits of downsizing images are fairly obvious (fewer pixels = fewer operations). Because recoloring the images simplifies their detail anyways, downsizing prior to recoloring doesn't run a very high risk of losing important information. A general guideline for resizing is that any distinguishable features of interest should still take up at least 2 pixels (preferably with a margin of error) in the resized image.

Value

An object of S3 class `recolorize` with the following attributes:

1. `original_img`: The original image, as a raster array.
2. `centers`: A matrix of color centers in RGB (0-1 range).
3. `sizes`: The number of pixels assigned to each color cluster.
4. `pixel_assignments`: A matrix of color center assignments for each pixel.
5. `call`: The call(s) used to generate the `recolorize` object.

Examples

```
# filepath to image
img <- system.file("extdata/chongi.png", package = "recolorize")

# default: histogram, 2 bins/channel
rc <- recolorize(img)

# we can also have different numbers of bins per channel
rc <- recolorize(img, bins = c(4, 1, 1)) # mostly red
rc <- recolorize(img, bins = c(1, 4, 1)) # mostly green
rc <- recolorize(img, bins = c(1, 1, 4)) # mostly blue

# kmeans can produce a better fit with fewer colors
rc <- recolorize(img, method = "kmeans", n = 8)

# increasing numbers of kmean colors
recolored_images <- setNames(vector("list", length = 10), c(1:10))
for (i in 1:10) {
  kmeans_recolor <- recolorize(img, method = "kmeans",
                               n = i)
}

# kmeans, 10 colors
kmeans_recolor <- recolorize(img, method = "kmeans",
                             n = 8, plotting = FALSE)
hist_recolor <- recolorize(img, method = "hist",
                           bins = 2, plotting = FALSE)
# to reset graphical parameters:
current_par <- graphics::par(no.readonly = TRUE)

# compare binning vs. kmeans clustering
layout(matrix(c(1, 2, 3), ncol = 3))
plot(kmeans_recolor$original_img); title("original")
plot(recoloredImage(kmeans_recolor, type = "raster")); title("kmeans")
plot(recoloredImage(hist_recolor, type = "raster")); title("binning")

graphics::par(current_par)
```

`recolorize2` *Recolorize with automatic thresholding*

Description

Calls recolorize and recluster in sequence, since these are often very effective in combination.

Usage

```
recolorize2(
  img,
  method = "histogram",
  bins = 2,
  n = 5,
  cutoff = 20,
  channels = 1:3,
  n_final = NULL,
  color_space = "sRGB",
  recluster_color_space = "Lab",
  refit_method = "impose",
  ref_white = "D65",
  lower = NULL,
  upper = NULL,
  transparent = TRUE,
  resize = NULL,
  rotate = NULL,
  plotting = TRUE
)
```

Arguments

`img`	Path to the image (a character vector) or a 3D image array as read in by `png::readPNG()` `{readImage}`.
`method`	Method for clustering image colors. One of either `histogram` or `kmeans`. See details.
`bins`	If `method` = `"histogram"`, either the number of bins per color channel (if a single number is provided) OR a vector of length 3 with the number of bins for each channel.
`n`	If `method` = `"kmeans"`, the number of color clusters to fit.
`cutoff`	Numeric similarity cutoff for grouping color centers together. The range is in absolute Euclidean distance. In CIE Lab space, it is greater than 0-100, but cutoff values between 20 and 80 will usually work best. In RGB space, range is 0-sqrt(3). See recluster details.
`channels`	Numeric: which color channels to use for clustering. Probably some combination of 1, 2, and 3, e.g., to consider only luminance and blue-yellow (b-channel) distance in CIE Lab space, channels = c(1, 3 (L and b).

`n_final`	Final number of desired colors; alternative to specifying a similarity cutoff. Overrides `similarity_cutoff` if provided.
`color_space`	Color space in which to minimize distances, passed to `[grDevices]{convertColor}`. One of "sRGB", "Lab", or "Luv". Default is "sRGB".
`recluster_color_space`	Color space in which to group colors for reclustering. Default is CIE Lab.
`refit_method`	Method for refitting the image with the new color centers. One of either "impose" or "merge". `imposeColors()` refits the original image using the new colors (slow but often better results). `mergeLayers()` merges the layers of the existing recolored image. This is faster since it doesn't require a new fit, but can produce messier results.
`ref_white`	Reference white for converting to different color spaces. D65 (the default) corresponds to standard daylight.
`lower`, `upper`	RGB triplet ranges for setting a bounding box of pixels to mask. See details.
`transparent`	Logical. Treat transparent pixels as background? Requires an alpha channel (PNG).
`resize`	A value between 0 and 1 for resizing the image (ex. `resize = 0.5` will reduce image size by 50%). Recommended for large images as it can speed up analysis considerably. See details.
`rotate`	Degrees to rotate the image clockwise.
`plotting`	Logical. Plot final results?

Value

An object of S3 class `recolorize` with the following attributes:

1. `original_img`: The original image, as a raster array.
2. `centers`: A matrix of color centers in RGB (0-1 range).
3. `sizes`: The number of pixels assigned to each color cluster.
4. `pixel_assignments`: A matrix of color center assignments for each pixel.
5. `call`: The call(s) used to generate the `recolorize` object.

See Also

recolorize, recluster

Examples

```
# get image path
img <- system.file("extdata/corbetti.png", package = "recolorize")

# fit recolorize:
rc <- recolorize2(img, bins = 2, cutoff = 45)
```

`recolorizeVector` *Convert a recolorize object to a vector*

Description

Converts a `recolorize` color map to a set of polygons, which can be plotted at any scale without losing quality (as opposed to the pixel-based bitmap format). Requires the `raster`, `rgeos`, and `sp` packages to be installed. Useful for creating nice visualizations; slow on large images. It's recommended to fit a `recolorize` object by reducing the original image first, rather than the `resize` argument here, which reduces the color map itself (to mixed results).

Usage

```
recolorizeVector(
  recolorize_obj,
  size_filter = 0.1,
  smoothness = 1,
  base_color = "default",
  plotting = FALSE,
  resize = 1,
  ...
)
```

Arguments

`recolorize_obj`	An object of class `recolorize`, as generated by recolorize, recolorize2, imposeColors, or wernerColor.
`size_filter`	The size (as a proportion of the shortest dimension of the image) of the color patch elements to absorb before vectorizing. Small details (e.g. stray pixels) tend to look very strange after vectorizing, so removing these beforehand can improve results.
`smoothness`	Passed to smoothr::smooth using the `"ksmooth"` method for smoothing the jagged lines that result from converting pixel coordinates to polygon vertices. Higher values = more smoothing.
`base_color`	The color to use to fill in the gaps that can result from smoothing. If `base_color = "default"`, defaults to the darkest color in the palette. Otherwise, should be the numeric index of one of the colors in `recolorize_obj$centers` to use.
`plotting`	Logical. Plot results while computing?
`resize`	Proportion by which to resize the color map before turning into a polygon, e.g. `resize = 0.5` will reduce color map size by 50%. Speeds up the function, but you will almost always get better results by resizing the initial image when fitting the `recolorize` object.
`...`	Plotting parameters, passed on to graphics::plot.

Details

Although vector objects will typically be smaller than `recolorize` objects, because they only need to specify the XY coordinates of the perimeters of each polygon, they can still be fairly large (and take a long time to calculate). Users can try a few things to speed this up: using lower smoothness values; setting `plotting = FALSE`; resizing the image (preferably when fitting the initial `recolorize` object); and reducing the complexity of the color patches using absorbLayer or editLayer (e.g. by absorbing all components < 10 pixels in size). Still, expect this function to take several minutes on even moderately sized images–it takes about 7-10 seconds for the ~200x100 pixel images in the examples! Once the function finishes running, however, plotting is quite fast, and the objects themselves are smaller than the `recolorize` objects.

Value

A `vector_recolorize` object, which is a list with the following elements:

1. `base_layer`: The base polygon, essentially the image silhouette.
2. `layers`: A list of sp::SpatialPolygonsDataFrame polygons, one per color patch.
3. `layer_colors`: The colors (as hex codes) for each polygon.
4. `base_color`: The color (as hex code) for the base polygon.
5. `asp`: The original image aspect ratio, important for plotting.

Examples

```
img <- system.file("extdata/corbetti.png", package = "recolorize")
rc <- recolorize2(img, cutoff = 45)

# to reset graphical parameters:
current_par <- graphics::par(no.readonly = TRUE)

# takes ~10 seconds
as_vector <- recolorizeVector(rc, smoothness = 5,
                              size_filter = 0.05)

# to save as an SVG with a transparent background and
# no margins (e.g. for an illustration figure):
grDevices::svg("recolorize_vector.svg",
height = 4, width = 2, bg = "transparent")
par(mar = rep(0, 4))
plot(as_vector)
dev.off()

# and to avoid spamming your working directory, run this line to remove
# the file we just wrote:
file.remove("recolorize_vector.svg")

graphics::par(current_par)
```

`recolorize_adjacency` *Run* `pavo`*'s adjacency and boundary strength analysis on a* `recolorize` *object*

Description

Run adjacency (Endler 2012) and boundary strength (Endler et al. 2018) analysis directly on a `recolorize` object, assuming a human viewer (i.e. using CIE Lab and HSL color distances that correspond to perceptual distances of human vision). This is achieved by converting the `recolorize` object to a pavo::classify object, converting the colors to HSL space, and calculating a pavo::coldist object for CIE Lab color space before running pavo::adjacent.

Usage

```
recolorize_adjacency(
  recolorize_obj,
  xscale = 1,
  coldist = "default",
  hsl = "default",
  ...
)
```

Arguments

`recolorize_obj`	A `recolorize` object.
`xscale`	The length of the x-axis, in preferred units. Passed to pavo::adjacent.
`coldist`	A pavo::coldist object; otherwise, this argument is ignored and a `coldist` object for human vision is calculated from RGB colors converted to CIE Lab using cielab_coldist.
`hsl`	A dataframe with `patch`, `hue`, `sat` and `lum` columns specifying the HSL values for each color patch, to be passed to pavo::adjacent. Otherwise, this argument is ignored and HSL values are calculated for human vision from the RGB colors in the `recolorize` object.
`...`	Further arguments passed to pavo::adjacent.

Details

Eventually, the plan is to incorporate more sophisticated color models than using human perceptual color distances, i.e. by allowing users to match color patches to spectra. However, this does return reasonable and informative results so long as human vision is an appropriate assumption for the image data.

Value

The results of pavo::adjacent; see that documentation for the meaning of each specific value.

See Also

pavo::adjacent, classify_recolorize

Examples

```
img <- system.file("extdata/chongi.png", package = "recolorize")
recolorize_obj <- recolorize(img, method = "k", n = 2)
recolorize_adjacency(recolorize_obj)
```

`recolorize_to_patternize`
Convert a recolorize object to a raster object

Description

Convert from a `recolorize` object to a list of RasterLayer objects, the format required by the `patternize` package. Note that most of the downstream `patternize` functions that require lists of RasterLayer objects mostly require lists of these lists, so you will probably need to use this function on a list of `recolorize` objects.

Usage

```
recolorize_to_patternize(recolorize_obj, return_background = FALSE)
```

Arguments

`recolorize_obj`	A `recolorize` object.
`return_background`	Logical.

Details

Note that this function does not retain the colors of the layers – you won't be able to convert back to a recolorize object from this object.

Value

A list of RasterLayer objects, one per color class.

Examples

```
# fit recolorize object:
img <- system.file("extdata/ephippigera.png", package = "recolorize")
rc <- recolorize2(img)

# takes ~10 sec to run:
```

```
# convert to a raster list:
as_raster_list <- recolorize_to_patternize(rc)
```

`recolorize_to_png` *Save a recolored image as a PNG*

Description

Saves a recolored image from a recolorize object to a PNG. This is done by calling recoloredImage and png::writePNG.

Usage

```
recolorize_to_png(recolorize_obj, filename = "")
```

Arguments

`recolorize_obj`	A recolorize object.
`filename`	Filename for saving the PNG.

Details

This function saves a png with the same dimensions (in pixels) as the image that was originally provided to recolorize (meaning if you resized your original image, the resulting PNG will also be smaller). Anything more complicated can be created with custom scripts: for example, you could create a vector image using recolorizeVector, and then save this as a PNG of any resolution/size.

Value

No return value; saves a PNG file to the specified location.

Examples

```
img <- system.file("extdata/corbetti.png", package = "recolorize")
rc <- recolorize2(img, cutoff = 45)

# save a PNG:
recolorize_to_png(rc, "corbetti_recolored.png")

# remove the PNG (so this example doesn't spam your working directory)
file.remove("corbetti_recolored.png")
```

`reorder_colors` *Reorder colors in a recolorize object*

Description

Often for batch processing purposes, it is important to ensure that color centers fit using different methods are in the same order.

Usage

```
reorder_colors(recolorize_obj, col_order, plotting = FALSE)
```

Arguments

`recolorize_obj`	An object of class `recolorize`.
`col_order`	A numeric vector of the length of the number of color centers in the `recolorize` object specifying the order of the colors.
`plotting`	Logical. Plot the results?

Details

While you can manually specify the `col_order` vector, one way to automatically order the colors according to an external color palette (as might be needed for batch processing) is to use the match_colors function, although it is recommended to double-check the results.

Value

A `recolorize` object.

Examples

```
img <- system.file("extdata/corbetti.png", package = "recolorize")
rc <- recolorize2(img, cutoff = 45)
ref_palette <- c("mediumblue", "olivedrab", "tomato2", "beige", "grey10")
col_order <- match_colors(ref_palette, rc$centers, plotting = TRUE)
rc2 <- reorder_colors(rc, col_order, plotting = FALSE)

# the colors are reordered, but not changed to match the reference palette:
plot(rc2)

# you can also change them to the reference palette:
rc2$centers <- t(grDevices::col2rgb(ref_palette) / 255)
plot(rc2)
```

`rerun_recolorize` *Rerun the sequence of calls used to produce a recolorize object*

Description

Evaluates the series of calls in the 'call' element of a recolorize object, either on the original image (default) or on another image. It will almost always be easier (and better practice) to define a new function that calls a series of recolorize function in order than to use this function!

Usage

```
rerun_recolorize(recolorize_obj, img = "original")
```

Arguments

`recolorize_obj`	An object of S3 class 'recolorize'.
`img`	The image on which to call the recolorize functions. If left as "original" (the default), functions are called on the original image stored in the recolorize object. Otherwise can be an object taken by the `img` argument of recolorize functions (a path to an image or an image array).

Details

This function utilizes `eval` statements to evaluate the calls that were stored in the `call` element of the specified recolorize object. This makes it potentially more unpredictable than simply defining your own function, which is preferable.

Value

A `recolorize` object.

Examples

```
# list images
corbetti <- system.file("extdata/corbetti.png", package = "recolorize")
chongi <- system.file("extdata/chongi.png", package = "recolorize")

# fit a recolorize object by running two functions in a row:
rc <- recolorize(corbetti, bins = 2, plotting = FALSE)
rc <- recluster(rc, cutoff = 45)

# check out the call structure (a list of commands that were run):
rc$call

# we can rerun the analysis on the same image (bit pointless):
rerun <- rerun_recolorize(rc)

# or, we can rerun it on a new image:
rerun_chongi <- rerun_recolorize(rc, img = chongi)
```

`rgb2hsl` *Convert RGB colors to HSL*

Description

Convert RGB colors (0-1 range) to HSL (hue-saturation-luminance) space. Used for passing RGB colors to pavo::adjacent.

Usage

```
rgb2hsl(rgb_matrix, radians = TRUE, pavo_hsl = TRUE)
```

Arguments

`rgb_matrix`	RGB colors in an nx3 matrix (rows = colors, columns = channels).
`radians`	Logical. Return HSL colors in units of radians (`TRUE`) or degrees (`FALSE`)?
`pavo_hsl`	Logical. Return HSL matrix in a format that can be passed directly to pavo::adjacent as the `hsl` parameter?

Value

A dataframe with `patch`, `hue`, `sat`, and `lum` columns and one row per color (if `pavo_hsl` = `TRUE`) or a matrix of the HSL coordinates (if `pavo_hsl` = `FALSE`).

`splitByColor` *Split color clusters in a recolorize object into layers*

Description

Separates color clusters from a `recolorize()`, `recluster()`, or `imposeColors()` object into binary masks.

Usage

```
splitByColor(
  recolorize_obj,
  layers = "all",
  plot_method = c("overlay", "binary", "colormask", "none")
)
```

Arguments

`recolorize_obj`	A recolorize object from `recolorize()`, `recluster()`, or `imposeColors()`.
`layers`	Either `"all"` or a numeric vector of which color centers to return.
`plot_method`	Plotting method for plotting the color layers. Options are`"overlay"`, `"binary"`, `"colormask"`, or `"none"`.

Value

A list of binary matrices (1/white = color presence, 0/black = color absence), one per color center.

Examples

```
# get original fit
corbetti <- system.file("extdata/corbetti.png", package = "recolorize")
recolored_corbetti <- recolorize::recolorize(corbetti, plotting = TRUE)

# to reset graphical parameters:
current_par <- graphics::par(no.readonly = TRUE)

# make a layout
layout(matrix(c(1, 1:9), nrow = 2))
par(mar = c(0, 0, 2, 0))
# plot original
plotImageArray(recolored_corbetti$original_img)

# plot layers
corbetti_layers <- splitByColor(recolored_corbetti, plot_method = "over")

# plot binary maps
plotImageArray(recolored_corbetti$original_img)
for (i in 1:length(corbetti_layers)) {
  plotImageArray(corbetti_layers[[i]])
}

graphics::par(current_par)
```

`thresholdRecolor` *Drop minor colors from a recolorize object*

Description

Drops color patches whose cumulative sum (as a proportion of total pixels assigned) is equal to or less than `pct`, so that only the dominant color patches remain, and refits the object with the reduced set of color centers Useful for dropping spurious detail colors.

Usage

```
thresholdRecolor(recolorize_obj, pct = 0.05, plotting = TRUE, ...)
```

Arguments

`recolorize_obj`	An object of class `recolorize`.
`pct`	The proportion cutoff (0-1) for dropping color patches. The higher this value is, the more/larger color centers will be dropped.
`plotting`	Logical. Plot the results?
`...`	Further arguments passed to imposeColors, which is called for refitting a new recolorize object for the reduced set of clusters.

Details

This function is fairly simple in execution: the color centers are arranged by their sizes, largest to smallest, and their cumulative sum is calculated. The minimum number of color centers to reach a cumulative sum equal to or greater than the cutoff (`1 - pct`) is retained, and these dominant colors are used to re-fit the image. Despite being straightforward, this can be a surprisingly useful function.

Value

A `recolorize` object.

Examples

```
img <- system.file("extdata/fulgidissima.png", package = "recolorize")
init_fit <- recolorize(img, bins = 3)
thresh_fit <- thresholdRecolor(init_fit, pct = 0.1)

# if you take it too far, you just get one color back:
thresh_fit_oops <- thresholdRecolor(init_fit, pct = 1)
```

`werner` *Werner's nomenclature of colors*

Description

A table of the 110 colors described in "Werner's Nomenclature of Colors", the 1821 color reference by Patrick Syme (building on work by Abraham Gottlob Werner), notably used by Charles Darwin. Colors represent the average pixel color of each scanned swatch.

Usage

```
werner
```

Format

A data frame with 110 rows and 13 variables:

index The color index.

family The broad color category (white, red, etc).

name The original color name.

hex Color hex code.

Source

https://www.c82.net/werner/#colors

`wernerColor` *Remap an image to Werner's nomenclature*

Description

Remaps a recolorize object to the colors in Werner's Nomenclature of Colors by Patrick Syme (1821), one of the first attempts at an objective color reference in western science, notably used by Charles Darwin.

Usage

```
wernerColor(
  recolorize_obj,
  which_img = c("original", "recolored"),
  n_colors = 5
)
```

Arguments

`recolorize_obj`	A recolorize object as returned by `recolorize()`, `recluster()`, or `imposeColors()`.
`which_img`	Which image to recolor; one of either "original" or "recolored".
`n_colors`	Number of colors to list out in plotting, in order of size. Ex: `n_colors` = 5 will plot the 5 largest colors and their names. All colors are returned as a normal recolorize object regardless of `n_colors`; this only affects the plot.

Details

See https://www.c82.net/werner/ to check out the original colors.

Value

A `recolorize` object with an additional list element, `werner_names`, listing the Werner color names for each center.

Examples

```
# get an initial fit:
corbetti <- system.file("extdata/corbetti.png", package = "recolorize")
recolored_corbetti <- recolorize(corbetti, plotting = FALSE)

# recolor original image
corbetti_werner <- wernerColor(recolored_corbetti,
                               which_img = "original",
                               n_colors = 6)

# we can simplify the colors and then do it again:
corbetti_recluster <- recluster(recolored_corbetti,
                                cutoff = 45,
                                plot_hclust = FALSE)
corbetti_werner <- wernerColor(corbetti_recluster,
                               which_img = "recolored")
```

Index