Package 'quadtree'

June 29, 2021

Type Package

Title Quadtree Representation of Rasters

Version 0.0.0.9000

Date 2021-01-26

Description Provides a C++ implementation of a quadtree data structure. Functions are provided for creating a quadtree from a raster, with the level of 'coarseness' of the quadtree being determined by a user-supplied parameter. In addition, functions are provided to extract values of the quadtree at point locations. Functions for calculating least-cost paths between two points are also provided.

License GPL (>= 2)

Imports Rcpp (>= 1.0.5),
raster,
dplyr

LinkingTo Rcpp

RoxygenNote 7.1.1

NeedsCompilation yes

URL https://gitlab.com/dafriend/quadtree

R topics documented:

quadtree-package *Quadtree Representation of Rasters*

Description

Provides a C++ implementation of a quadtree data structure. Functions are provided for creating a quadtree from a raster, with the level of "coarseness" of the quadtree being determined by a user-supplied parameter. In addition, functions are provided to extract values of the quadtree at point locations. Functions for calculating least-cost paths between two points are also provided.

Details

To get an understanding of the most important aspects of the package, read the following:

1. For details on how a quadtree is constructed, see `qt_create`.
2. For details on the least-cost path functionality, see `qt_lcp_finder`.

Function summary:

`qt_create`: create a quadtree from a raster

`qt_extent`, `qt_extent_orig`: get the extent of a quadtree

`qt_extract`: extract values from the quadtree at point locations

`qt_lcp_finder`, `qt_find_lcp`: find the LCP between two points using the quadtree as a cost surface

`qt_plot`: plot a quadtree

`qt_proj4string`: get the proj4string of a quadtree

`qt_read`, `qt_write`: read and write a quadtree object to a file

Author(s)

Derek Friend

add_legend *Add a gradient legend to a plot*

Description

Adds a gradient legent to a plot

Usage

```
add_legend(
  zlim,
  col,
  lgd_box_col = NULL,
  lgd_x_pct = 0.5,
  lgd_y_pct = 0.5,
  lgd_wd_pct = 0.5,
```

```
  lgd_ht_pct = 0.5,
  bar_box_col = "black",
  bar_wd_pct = 0.2,
  bar_ht_pct = 1,
  ticks = NULL,
  ticks_n = 5,
  ticks_x_pct = 1
)
```

Arguments

`zlim`	two-element numeric vector; required; the min and max value of z
`col`	character vector; required; the colors that will be used to create the color ramp used in the plot.
`lgd_box_col`	character; color of the box to draw around the entire legend. If `NULL` (the default), no box is drawn
`lgd_x_pct`	numeric; location of the center of the legend in the x-dimension, as a fraction (0 to 1) of the *right margin area*, **not** the entire width
`lgd_y_pct`	numeric; location of the center of the legend in the y-dimension, as a fraction (0 to 1). Unlike `lgd_x_pct`, this **is** relative to the entire figure height (since the right margin area spans the entire vertical dimension)
`lgd_wd_pct`	numeric; width of the entire legend, as a fraction (0 to 1) of the right margin width
`lgd_ht_pct`	numeric; height of the entire legend, as a fraction (0 to 1) of the figure height
`bar_box_col`	character; color of the box to draw around the color bar. If `NULL`, no box is drawn
`bar_wd_pct`	numeric; width of the color bar, as a fraction (0 to 1) of the width of the *legend area* (**not** the entire right margin width)
`bar_ht_pct`	numeric; height of the color bar, as a fraction (0 to 1) of the height of the *legend area* (**not** the entire right margin height)
`ticks`	numeric vector; the z-values at which to place tick marks. If `NULL` (the default), tick placement is automatically calculated
`ticks_n`	integer; the number of ticks desired - only used if `ticks` is `NULL`. Note that this is an *approximate* number - the `pretty()` function from `grDevices` is used to generate "nice-looking" values, but it doesn't guarantee a set number of tick marks
`ticks_x_pct`	numeric; the x-placement of the tick labels as a fraction (0 to 1) of the width of the legend area. This corresponds to the *right-most* part of the text - i.e. a value of 1 means the text will end exactly at the right border of the legend area

Details

I took an HTML/CSS-like approach to determining the positioning - that is, each space is treated as -like space, and the position of objects within that space happens *relative to that space* rather then the entire space. The parameters prefixed by `lgd` are all relative to the right margin space and correspond to the box that contains the entire legend. The parameters prefixed `bar` and `ticks` are relative to the space within the legend box.

I obviously wrote this for plotting the quadtree, but there's nothing quadtree-specific about this particular function.

This function is used within qt_plot, so the user shouldn't call this function to manually create the legend. Customizations to the legend can be done via the legend_args parameter of qt_plot(). Using this function to plot the legend after using qt_plot() raises the possibility of the legend not corresponding correctly with the plot, and thus should be avoided.

Examples

```
set.seed(23)
mat = matrix(runif(64,0,1), nrow=8)
qt = qt_create(mat, .75)

par(mar=c(5,4,4,5))
qt_plot(qt,legend=FALSE)
add_legend(range(mat), rev(terrain.colors(100)))
```

---

get_coords *Get the extent of the figure area in plot units (for one dimension)*

Description

Given the coordinate range of a single dimension in user units (par("usr")) and the coordinates of that same coordinate range as a fraction of the current figure region (par("plt")), calculates the extent of the entire figure area in user units.

Usage

```
get_coords_axis(usr, plt)

get_coords(usr, plt)
```

Arguments

usr	two-element (get_coords_axis) or four-element (get_coords) numeric vector; specifies the user coordinates of the plot region. Can be retrieved using par("usr"), and subscripts can be used to get only one dimension (for get_coords_axis - i.e par("usr")[1:2])
plt	two-element (get_coords_axis) or four-element (get_coords) numeric vector; specifies the coordinates of the plot region as fractions of the figure region. Can be retrieved using par("plt"), and subscripts can be used to get only one dimension (for get_coords_axis - i.e par("plt")[1:2])

Details

get_coords_axis() is used to find the user coordinates of a single dimension of the figure area. In this case, usr and plt should both be two-element vectors corresponding to the same dimension (see examples). Both vectors need to be in the format c(max,min).

get_coords() is simply a wrapper for get_coords that does both dimensions at once. In this case the output of par("usr") and par("plt") can be directly supplied to the usr and plt parameters, respectively. Note that for both parameters the vectors must have length 4 and be in this order: c(xmin,xmax,ymin,ymax).

These functions were written for use in add_legend. In order to properly place the legend, I needed to know the extent of the entire figure region in user coordinates. However, there's nothing about this function that is specific to that one application, and could be used in other situations as well.

Understanding what these functions do (and why they're necessary) requires an understanding of the graphical parameters, and in particular what usr and plt represent. See ?par for more on these parameters.

See Also

Run ?par for more details on the usr and plt parameters

Examples

```
p = par() # retrieve the graphical parameters as a list
get_coords_axis(p$usr[1:2], p$plt[1:2]) # x-axis
get_coords_axis(p$usr[3:4], p$plt[3:4]) # y-axis

get_coords(p$usr, p$plt) # both dimensions at once
get_coords(par("usr"), par("plt")) #this also works
```

---

qt_create *Create a quadtree from gridded data*

Description

Create a quadtree from gridded data

Usage

```
qt_create(
  x,
  range_limit,
  max_cell_length = NULL,
  adj_type = "expand",
  resample_n_side = NULL,
  extent = NULL,
  proj4string = NULL
)
```

Arguments

x	a raster or a matrix. If x is a matrix, the extent and proj4string parameters can be used to set the extent and projection of the quadtree. If x is a raster, the extent and projection are derived from the raster.
range_limit	numeric; if the cell values within a quadrant have a range larger than this value, the quadrant is split. See 'Details' for more
max_cell_length	double; the maximum size allowed for a quadtree cell. If NULL no restrictions are placed on the quadtree cell size. See 'Details' for more
adj_type	character; either 'expand' or 'resample'. See 'Details' for more.

`resample_n_side`
: integer; if `adj_type` is `'expand'`, this number is used to determine the dimensions to resample the raster to

`extent`
: `Extent` object or else a four-element numeric vector describing the extent of the data (in this order: xmin, xmax, ymin, ymax). Only used when `x` is a matrix - this parameter is ignored if `x` is a raster. If no value is provided and `x` is a matrix, the extent is assumed to be `c(0,ncol(x),0,nrow(x))`.

`proj4string`
: character; proj4string describing the projection of the data. Only used when `x` is a matrix - this parameter is ignored if `x` is a raster. If no value is provided and `x` is a matrix, the 'proj4string' of the quadtree is set to `NA`.

Details

A quadtree is created from a raster by successively dividing the raster/matrix into smaller and smaller cells, with the decision on whether to divide a cell determined by `range_limit`. Initially, all of the cells in the raster are considered. If the difference between the maximum and minimum cell values exceeds `range_limit`, the raster is divided into four quadrants - otherwise, the raster is not divided further and the mean of all values in the raster is taken as the value for the resulting cell. Then, the process is repeated for each of those 'child' cells, and then for their children, and so on and so forth, until either `range_limit` is not exceeded or the smallest possible cell size has been reached.

If a quadrant contains both NA cells and non-NA cells, that quadrant is automatically divided. However, if a quadrant consists entirely of NA cells, that cell is not divided further (even if the cell is larger than `max_cell_length`).

If a given quadrant has dimensions that are not divisible by 2 (for example, 5x5), then the process stops. Because of this, only rasters that have dimensions that are a power of 2 can be divided down to their smallest cell size. In addition, the rasters should be square.

If `max_cell_length` is not `NA`, then the maximum cell size in the resulting quadtree will be `max_cell_length`. This essentially forces any quadrants larger than `max_cell_length` to split. The one exception is that a quadrant that contains entirely `NA` values will not be split.

To create quadtrees from rasters that have dimensions that are not a power of two and are not square, two options are provided. The choice of method is determined by the `adj_type` parameter.

In the 'expand' method, NA cells are added to the raster in order to create an expanded raster whose dimensions are a power of 2. The smallest number that is a power of two but greater than the larger dimension is used as the dimensions of the expanded raster. For example, if a raster has dimensions 546 x 978, NA cells are added to the top and right of the raster in order to create a raster with dimensions 1024 x 1024 (as 1024 is the smallest power of 2 that is also greater than 978).

In the 'resample' method, the raster is resampled in order to create a square matrix with dimensions that are a power of two. There are two steps. First, the raster must be made square. This is done in a way similar to the method described above. The smaller dimension is padded with NA cells in order to equal the larger dimension. For example, if the raster has dimensions 546 x 978, NA rows are added in order to create a raster with dimensions 978 x 978. In the second step, this raster is then resampled to a user-specified dimension (determined by the `resample_n_side` parameter). For example, the user could set `resample_n_side` to be 1024, which will resample the 978 x 978 raster to 1024 x 1024. This raster can then be used to create a quadtree. The dimensions should be a power of 2 (see above for an explanation), although other numbers will be accepted (but will trigger a warning).

Examples

```
#create raster of random values
```

```
nrow = 57
ncol = 75
rast = raster(matrix(runif(nrow*ncol), nrow=nrow, ncol=ncol), xmn=0, xmx=ncol, ymn=0, ymx=nrow)

#create quadtree using the 'expand' method - automatically adds NA cells to
#bring the dimensions to 128 x 128 before creating the quadtree
qt1 = qt_create(rast, range_limit = .9, adj_type="expand")
qt_plot(qt1) #plot the quadtree
qt_plot(qt1, crop=TRUE) #we can use 'crop=TRUE' if we don't want to see the padded NA's

#create quadtree using the 'resample' method - we'll resample to 128 since it's a power of 2
qt2 = qt_create(rast, range_limit = .9, adj_type="resample", resample_n_side = 128)
qt_plot(qt2)
qt_plot(qt2, crop=TRUE)

#now use the 'max_cell_length' argument to force any cells with sides longer
#than 2 to split
qt3 = qt_create(rast, range_limit = .9, max_cell_length = 2, adj_type="expand")

#compare qt1 (no max cell length) and qt3 (max cell length = 2)
par(mfrow=c(1,2))
qt_plot(qt1,crop=TRUE, main="no max cell length")
qt_plot(qt3,crop=TRUE, main="max cell length = 2")
```

`qt_extent` *Get the extent of a quadtree*

Description

Gets the extent of the quadtree as an 'extent' object (from the raster package)

Usage

```
qt_extent(quadtree)

qt_extent_orig(quadtree)
```

Arguments

`quadtree`	quadtree object

Details

`qt_extent` returns the total extent covered by the quadtree.

`qt_extent_orig` returns the extent of the original raster used to create the quadtree, before NA rows/columns were added to pad the dimensions. This essentially represents the extent in which the non-NA data occurs.

Value

Returns an 'extent' object

Examples

```
#create raster of random values
nrow = 57
ncol = 75
rast = raster(matrix(runif(nrow*ncol), nrow=nrow, ncol=ncol), xmn=0, xmx=ncol, ymn=0, ymx=nrow)
qt = qt_create(rast, .9, adj_type="expand")

qt_extent(qt)
qt_extent_orig(qt)
```

---

qt_extract *Extract the values of a quadtree at the given locations*

Description

Extract the cell values and optionally the cell extents

Usage

```
qt_extract(quadtree, pts, extents = FALSE)
```

Arguments

quadtree	A quadtree object
pts	A two-column matrix representing point coordinates. First column contains the x-coordinates, second column contains the y-coordinates
extents	boolean; if FALSE, a vector containing cell values is returned. If TRUE, a matrix is returned providing each cell's extent in addition to its value

Value

The return type depends on the value of extents.

If extents = FALSE, the function returns a numeric vector corresponding to the values at the points represented by pts. If a point falls within the quadtree extent and the corresponding cell is NA, NA is returned. If the point falls outside of the quadtree extent, NaN is returned.

If extents = TRUE, the function returns a 5-column numeric matrix providing the extent of each cell along with the cell's value. The 5 columns are, in this order: xmin, xmax, ymin, ymax, value. If a point falls in a NA cell, the cell extent is still returned but value will be NA. If a point falls outside of the quadtree, all values will be NaN.

Examples

```
# create raster of random values
nrow = 57
ncol = 75
rast = raster(matrix(runif(nrow*ncol), nrow=nrow, ncol=ncol), xmn=0, xmx=ncol, ymn=0, ymx=nrow)

# create quadtree
qt1 = qt_create(rast, range_limit = .9, adj_type="expand")

# create points at which we'll extract values
```

```
pts = cbind(-5:15, 45:65)

# plot the quadtree and the points
qt_plot(qt1, border_col="gray60")
points(pts, pch=16,cex=.6)

# extract values only
qt_extract(qt1,pts)

# extract the cell extents in addition to the values
qt_extract(qt1,pts,extents=TRUE)
```

`qt_lcp_finder` *Find the LCP between two points on a quadtree*

Description

Finds the least cost path (LCP) between two points, using a quadtree as a resistance surface

Usage

```
qt_lcp_finder(quadtree, start_point, xlims = NULL, ylims = NULL)

qt_find_lcp(lcp_finder, end_point, use_original_end_points = FALSE)
```

Arguments

`quadtree`	a quadtree object to be used as a resistance surface
`start_point`	numeric vector with 2 elements - the x and y coordinates of the starting point of the path(s)
`xlims`	numeric vector with 2 elements - paths will be constrained so that all points fall within the min and max x coordinates specified in `xlims`. If `NULL` the x limits of `quadtree` are used
`ylims`	same as `xlims`, but for y
`lcp_finder`	the LCP finder object returned from `qt_lcp_finder`
`end_point`	numeric vector with two elements - the x and y coordinates of the the destination point
`use_original_end_points`	boolean; by default the start and end points of the returned path are not the points given by the user but instead the centroids of the cells that those points fall in. If this parameter is set to `TRUE` the start and end points (representing the cell centroids) are replaced with the actual points specified by the user. Note that this is done after the calculation and has no effect on the path found by the algorithm.

Details

These two functions are intended to be used in conjunction with one another. `qt_lcp_finder` creates the object used to find the LCP(s), and `qt_find_lcp` uses this object to find the LCP to a given point. See 'Examples' for examples of its usage.

The code is structured this way to minimize the computation needed to compute multiple LCPs from a single point. Dijkstra's algorithm iteratively searches for least cost paths, and in the process of finding one LCP it inevitably finds LCPs to other points. Because of this, we can save computation if we use an object that can save its current state rather than having to replicate our work.

The LCP finder stops its search once it reaches the desired node. However, because state is saved, if another LCP is requested and the LCP to that point has not yet been calculated, the LCP finder starts where it left off rather than starting over again. If the LCP to that point has been calculated, it returns the path without having to perform the algorithm again.

Note, however, that this only applies to the LCPs found using the same starting point. If a different starting point is used, a different LCP finder object is needed.

As mentioned before, Dijkstra's algorithm is used to compute the shortest path. Dijkstra's algorithm is a network algorithm. The network used in this case consists of the cell centroids (nodes) and the neighbor connections (edges). The cost of each edge is taken as the length of the edge times the weight - because the edge travels between two cells, the cost of the edge is weighted by the distance that falls within each cell.

Because of the heterogeneous nature of a quadtree, the paths found likely won't reflect the 'true' least cost path. This is because treating the centroids of the cells as the nodes introduces some distortion, especially with large cells.

Note that the `xlims` and `ylims` arguments in `qt_lcp_finder` can be used to restrict the search space to the rectangle defined by `xlims` and `ylims`. This speeds up the computation of the LCP by limiting the number of cells considered.

Value

`qt_lcp_finder` returns an LCP finder object. If `start_point` falls outside of the quadtree extent, `NULL` is returned.

`qt_find_lcp` returns a five column matrix representing the least cost path. It has the following columns:

- `x`: x coordinate of this point
- `y`: y coordinate of this point
- `cost_tot`: the cumulative cost up to this point
- `dist_tot`: the cumulative distance up to this point - note that this is not straight-line distance, but instead the distance along the path
- `cost_cell`: the cost of the cell that contains this point

If no path is possible between the two points, a 0-row matrix with the previously described columns is returned. Also, note that when creating the LCP finder object using `qt_lcp_finder`, `NULL` will be returned if `start_point` falls outside of the quadtree. If `NULL` is passed to the `lcp_finder` parameter, a 0-row matrix is returned.

IMPORTANT NOTE: the `use_original_end_points` options ONLY changes the x and y coordinates of the first and last points - it doesn't change the `cost_tot` or `dist_tot` columns. This means that even though the start and end points have changed, the `cost_tot` and `dist_tot` columns still represent the cost and distance using the cell centroids of the start and end cells.

Examples

```
# create raster of random values
nrow = 57
ncol = 75
set.seed(4)
rast = raster(matrix(runif(nrow*ncol), nrow=nrow, ncol=ncol), xmn=0, xmx=ncol, ymn=0, ymx=nrow)

# create quadtree
qt1 = qt_create(rast, range_limit = .9, adj_type="expand")
qt_plot(qt1,crop=TRUE)
start_pt = c(.231,.14)
end_pt = c(74.89,56.11)
# create the LCP finder object
spf = qt_lcp_finder(qt1, start_pt)

# use the LCP finder object to find the LCP to a certain point
# this path will have the cell centroids as the start and end points
path1 = qt_find_lcp(spf, end_pt)
# this path will be identical to path1 except that the start and end points
# will be the user-provided start and end points rather than the cell centroids
path2 = qt_find_lcp(spf, end_pt, use_original_end_points = TRUE)

head(path1)
head(path2)

# plot the result
qt_plot(qt1, crop=TRUE, border_col="gray60")
points(rbind(start_pt, end_pt), pch=16, col="red")
lines(path1[,1:2], col="black", lwd=2.5)
lines(path2[,1:2], col="red", lwd=1)
points(path1, cex=.7, pch=16)

#-------------------
# a larger example to demonstrate run time
#-------------------
nrow = 570
ncol = 750
rast = raster(matrix(runif(nrow*ncol), nrow=nrow, ncol=ncol), xmn=0, xmx=ncol, ymn=0, ymx=nrow)

qt1 = qt_create(rast, range_limit = .9, adj_type="expand")
spf = qt_lcp_finder(qt1, c(1,1))

# the LCP finder saves state. So finding the path the first time requires
# computation, and takes longer, but running it again is nearly instantaneous
system.time(qt_find_lcp(spf, c(740,560))) #takes longer
system.time(qt_find_lcp(spf, c(740,560))) #runs MUCH faster

# in addition, because of how Dijkstra's algorithm works, the LCP finder also
# found many other LCPs in the course of finding the first LCP, meaning that
# subsequent LCP queries for different destination points will be much faster
# (since the LCP finder saves state)
system.time(qt_find_lcp(spf, c(740,1)))
system.time(qt_find_lcp(spf, c(1,560)))

# now save the paths so we can plot them
path1 = qt_find_lcp(spf, c(740,560))
```

```
path2 = qt_find_lcp(spf, c(740,1))
path3 = qt_find_lcp(spf, c(1,560))

qt_plot(qt1, crop=TRUE, border_col="transparent")
lines(path1[,1:2])
lines(path2[,1:2], col="red")
lines(path3[,1:2], col="blue")
```

`qt_plot` *Plot a quadtree object*

Description

Plot a quadtree object

Usage

```
qt_plot(
  qt,
  colors = NULL,
  nb_line_col = NULL,
  border_col = "black",
  xlim = NULL,
  ylim = NULL,
  crop = FALSE,
  na_col = "white",
  adj_mar_auto = 6,
  legend = TRUE,
  legend_args = list(),
  ...
)
```

Arguments

`qt`	a `quadtree` object
`colors`	character vector; the colors that will be used to create the color ramp used in the plot. If no argument is provided, `terrain.colors(100,rev=TRUE)` is used.
`nb_line_col`	character; the color of the lines drawn between neighboring cells. If `NULL` (the default), these lines are not plotted
`border_col`	character; the color to use for the cell borders. Use 'transparent' if you don't want borders to be shown
`xlim`	two element numeric vector; defines the minimum and maximum values of the x axis.
`ylim`	two element numeric vector; defines the minimum and maximum values of the y axis.
`crop`	boolean; if `TRUE`, only displays the extent of the original raster, thus ignoring any of the NA cells that were added to pad the raster before making the quadtree. Ignored if either `xlim` or `ylim` are non-`NULL`
`na_col`	character; the color to use for NA cells. If `NULL`, NA cells are not plotted

adj_mar_auto	numeric; if not NULL, it checks the size of the right margin (par("mar")[4]) - if it is less than the provided value and legend is TRUE, then it sets it to be the provided value in order to make room for the legend (after plotting, it resets it to its original value). Default is 6.
legend	boolean; if TRUE (the default) a legend is plotted in the right margin
legend_args	named list; contains arguments that are sent to the add_legend function. See the help page for add_legend for the parameters. Note that the two required parameters to add_legend, zlim and cols, are supplied automatically, so if the list contains elements named zlim or cols, they will be ignored.
...	arguments passed to the default plot function

Examples

```
# create raster of random values
nrow = 57
ncol = 75
set.seed(2)
rast = raster(matrix(runif(nrow*ncol), nrow=nrow, ncol=ncol), xmn=0, xmx=ncol, ymn=0, ymx=nrow)

# create quadtree
qt1 = qt_create(rast, range_limit = .9, adj_type="expand")

# -----------
# DEFAULT
# -----------

# default - no additional parameters provided
qt_plot(qt1)

# -----------
# CHANGE PLOT EXTENT
# -----------

# note that additional parameters like 'main', 'xlab', 'ylab', etc. will be
# passed to the default 'plot()' function

# crop extent to the original extent of the raster
qt_plot(qt1, crop=TRUE, main="cropped")

# use 'xlim' and 'ylim' to zoom in on an area
qt_plot(qt1, xlim = c(30,50), ylim = c(10,20), main="zoomed in")

# -----------
# COLORS
# -----------

# change border color
qt_plot(qt1, border_col="transparent") #no borders
qt_plot(qt1, border_col="gray60")

# change color palette
qt_plot(qt1, colors=c("blue", "yellow", "red"))
qt_plot(qt1, colors=hcl.colors(100))
qt_plot(qt1, colors=c("black", "white"))
```

```
# change color of NA cells
qt_plot(qt1, na_col="pink")

# don't plot NA cells
qt_plot(qt1, na_col=NULL)

# -----------
# SHOW NEIGHBOR CONNECTIONS
# -----------

# plot all neighbor connections
qt_plot(qt1, nb_line_col="black", border_col="gray60")

# don't plot connections to NA cells
qt_plot(qt1, crop=TRUE, nb_line_col="black", border_col="gray60", na_col=NULL)

# -----------
# LEGEND
# -----------

# no legend
qt_plot(qt1, legend=FALSE)

# increase right margin size
qt_plot(qt1, adj_mar_auto=10)

# use 'legend_args' to customize the legend
qt_plot(qt1, adj_mar_auto=10, legend_args=list(lgd_ht_pct=.8, bar_wd_pct=.4))
```

`qt_proj4string` *Retrieve the proj4string of a quadtree*

Description

Retrieve the proj4string of a quadtree

Usage

```
qt_proj4string(quadtree)
```

Arguments

`quadtree`	a quadtree object

Value

A character containing the proj4string

qt_read *Read/write a quadtree*

Description

Read/write a quadtree

Usage

```
qt_read(filepath)

qt_write(quadtree, filepath)
```

Arguments

filepath	character; the filepath to read from or write to
quadtree	quadtree object; the quadtree to write

Details

To read/write a quadtree object, the C++ library `cereal` is used to serialize the quadtree and save it to a file. The file extension is unimportant - it can be anything (I've been using the extension '.qtree').

Note that typically the quadtree isn't particularly space-efficient- it's not uncommon for a quadtree file to be larger than the original raster file (although, of course, this depends on how 'coarse' the quadtree is in relation to the original raster). This is likely because the quadtree has to store much more information about each cell (the x and y limits, its value, pointers to its neighbors, among other things) while a raster can store only the value since the coordinates of the cell can be determined from the knowledge of the extent and the dimensions of the raster.

It's entirely possible that a quadtree implemention could be written that is MUCH more space efficient. However, this was not the primary goal when creating the quadtree.

Examples

```
qt = qt_read("path/to/quadtree.qtree")
qt_write(qt, "path/to/newQuadtree.qtree")
```

Index