Package: fasterRaster (via r-universe)

January 17, 2025

Type Package

Title Faster Raster and Spatial Vector Processing Using 'GRASS GIS'

Version 8.4.0.4

Date 2024-12-16

Maintainer Adam B. Smith <adam.smith@mobot.org>

Description Processing of large-in-memory/large-on disk rasters and spatial vectors using 'GRASS GIS' <https://grass.osgeo.org/>. Most functions in the 'terra' package are recreated. Processing of medium-sized and smaller spatial objects will nearly always be faster using 'terra' or 'sf', but for large-in-memory/large-on-disk objects, 'fasterRaster' may be faster. To use most of the functions, you must have the stand-alone version (not the 'OSGeoW4' installer version) of 'GRASS GIS' 8.0 or higher.

Depends R (>= 4.0.0)

Imports data.table (>= 1.14.8), DT, graphics, grDevices, methods, omnibus (>= 1.2.11), rgrass (>= 0.3-9), rpanel, sf, shiny, terra (>= 1.7), utils

Suggests knitr, rmarkdown

Roxygen list(markdown = TRUE)

License GPL (>=3)

SystemRequirements GRASS (>= 8)

URL https://github.com/adamlilith/fasterRaster, https://adamlilith.github.io/fasterRaster/

BugReports https://github.com/adamlilith/fasterRaster/issues

VignetteBuilder knitr

Encoding UTF-8

LazyData true

LazyLoad yes

RoxygenNote 7.3.2

Config/pak/sysreqs libgdal-dev gdal-bin libgeos-dev make libxml2-dev libssl-dev libproj-dev libsqlite3-dev libudunits2-dev zlib1g-dev

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

RemoteUrl https://github.com/adamlilith/fasterraster

RemoteRef HEAD

RemoteSha fc34c10ef5e0b9b0d6c925e18c061abbe8fedc22

Contents

`activeCat,GRaster-method`

Get or set the column with category labels in a categorical raster

Description

These functions return or set the column of the labels to be matched to each value in the raster of a categorical `GRaster` (see `vignette("GRasters", package = "fasterRaster")`). *Important*: Following `terra::activeCat()`, the first column in the "levels" table is ignored, so an "active category" value of 1 means the second column is used as labels, a value of 2 means the third is used, and so on.

- `activeCat()` returns the column of the labels to be matched to each value in the raster for a single raster layer.
- `activeCats()` does the same, but for all layers of a `GRaster`.
- `activeCat()<-` sets the column to be used as category labels.

Usage

```
## S4 method for signature 'GRaster'
activeCat(x, layer = 1, names = FALSE)

## S4 method for signature 'GRaster'
activeCats(x, names = FALSE)

## S4 replacement method for signature 'GRaster'
activeCat(x, layer = 1) <- value

## S4 replacement method for signature 'GRaster'
activeCat(x, layer = 1) <- value

## S4 replacement method for signature 'GRaster'
activeCat(x, layer = 1) <- value
```

Arguments

`x`	A categorical `GRaster`.
`layer`	Numeric, integer, logical, or character: Indicates for which layer(s) to get or set the active category column. This can be a number (the index of the raster(s)), a logical vector (`TRUE` ==> get/set the active category column, `FALSE` ==> leave as-is), or a character vector (names of layers).
`names`	Logical: If `TRUE`, display the name(s) of the active column(s). If `FALSE` (default), report the index of the active column. Following `terra::activeCat()`, the first column in the levels table is ignored. So, an active column of "1" means the second column is active. "2" means the third column is active, and so on.

value — Numeric, integer, or character. Following terra::activeCat(), the first column in each levels table is ignored. So, if you want the second column to be the category label, use 1. If you want the third column, use 2, and so on. You can also specify the active column by its column name (though this can't be the first column's name).

Value

`activeCat()` returns an integer or character of the active column index or name. `activeCats()` returns a vector of indices or names. `activeCat()<-` returns a `GRaster`.

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data: Land cover raster
madCover <- fastData("madCover")

# Convert categorical SpatRaster to categorical GRaster:
cover <- fast(madCover)

### Properties of categorical rasters

cover # note categories
is.factor(cover) # Is the raster categorical?
nlevels(cover) # number of levels
levels(cover) # just the value and active column
cats(cover) # all columns
minmax(cover) # min/max values
minmax(cover, levels = TRUE) # min/max categories
catNames(cover) # column names of the levels table
missingCats(cover) # categories in table with no values in raster
freq(cover) # frequency of each category (number of cells)
zonalGeog(cover) # geometric statistics

### Active column

# Which column sets the category labels?
activeCat(cover)
activeCat(cover, names = TRUE)

activeCats(c(cover, cover))

# Choose a different column for category labels:
levels(cover)
activeCat(cover) <- 2
levels(cover)

### Managing levels tables
```

```
# Remove unused levels:
nlevels(cover)
cover <- droplevels(cover)
nlevels(cover)

# Re-assign levels:
value <- c(20, 30, 40, 50, 120, 130, 140, 170)
label <- c("Cropland", "Cropland", "Forest", "Forest",
 "Grassland", "Shrubland", "Herbaceous", "Flooded")

newCats <- data.frame(value = value, label = label)

cover <- categories(cover, layer = 1, value = newCats)
cats(cover)

# This is the same as:
levels(cover) <- newCats
cats(cover)

# Are there any values not assigned a category?
missingCats(cover)

# Let's assign a category for value 210 (water):
water <- data.frame(value = 210, label = "Water")
addCats(cover) <- water
levels(cover)

# Add more information to the levels table using merge():
landType <- data.frame(
     Value = c(20, 30, 40, 50, 120),
     Type = c("Irrigated", "Rainfed", "Broadleaf evergreen",
     "Broadleaf deciduous", "Mosaic with forest")
)
cats(cover)
cover <- addCats(cover, landType, merge = TRUE)
cats(cover)

### Logical operations on categorical rasters

cover < "Forest" # 1 for cells with a value < 40, 0 otherwise
cover <= "Forest" # 1 for cells with a value < 120, 0 otherwise
cover == "Forest" # 1 for cells with value of 40-120, 0 otherwise
cover != "Forest" # 1 for cells with value that is not 40-120, 0 otherwise
cover > "Forest" # 1 for cells with a value > 120, 0 otherwise
cover >= "Forest" # 1 for cells with a value >= 120, 0 otherwise

cover %in% c("Cropland", "Forest") # 1 for cropland/forest cells, 0 otherwise

### Combine categories from different rasters

# For the example, will create a second categorical raster fromm elevation.
```

```
# Divide elevation raster into "low/medium/high" levels:
madElev <- fastData("madElev")
elev <- fast(madElev)
elev <- project(elev, cover, method = "near") # convert to same CRS
fun <- "= if(madElev < 100, 0, if(madElev < 400, 1, 2))"
elevCat <- app(elev, fun)

levs <- data.frame(
     value = c(0, 1, 2),
     elevation = c("low", "medium", "high")
)
levels(elevCat) <- list(levs)

# Combine levels:
combined <- concats(cover, elevCat)
combined
levels(combined)

# Combine levels, treating value/NA combinations as new categories:
combinedNA <- concats(cover, elevCat, na.rm = FALSE)
combinedNA
levels(combinedNA)

}
```

---

add<- *"Stack" a GRaster*

Description

This function "stacks" one `GRaster` with another. It has the same functionality as `c()`.

Usage

```
## S4 replacement method for signature 'GRaster,GRaster'
add(x) <- value
```

Arguments

`x`, `value`	A `GRaster`.

Value

A `GRaster`.

See Also

`c()`, terra::add<-, `terra::c()`

Examples

```
if (grassStarted()) {

# Setup
library(terra)

### GRasters

# Example data
madElev <- fastData("madElev") # elevation raster
madForest2000 <- fastData("madForest2000") # forest raster
madForest2014 <- fastData("madForest2014") # forest raster

# Convert SpatRasters to GRasters
elev <- fast(madElev)
forest2000 <- fast(madForest2000)
forest2014 <- fast(madForest2014)

# Re-assigning values of a GRaster
constant <- elev
constant[] <- pi
names(constant) <- "pi_raster"
constant

# Re-assigning specific values of a raster
replace <- elev
replace[replace == 1] <- -20
replace

# Subsetting specific values of a raster based on another raster
elevInForest <- elev[forest2000 == 1]
plot(c(elev, forest2000, elevInForest), nr = 1)

# Adding and replacing layers of a GRaster
rasts <- c(elev, constant, forest2000)

# Combine with another layer:
add(rasts) <- forest2014 # one way
rasts

rasts <- c(rasts, forest2014) # another way

### Subsetting GRaster layers

# Subset:
rasts <- c(elev, forest2000, forest2014)
rasts[[2:3]]
subset(rasts, 2:3)
subset(rasts, c("madForest2000", "madElev"))
rasts[[c("madForest2000", "madElev")]]
rasts$madForest2000
```

```
# Get every other layer:
rasts[[c(FALSE, TRUE)]]

### Replacing layers of a GRaster

# Replace a layer
logElev <- log(elev)
names(logElev) <- "logElev"
rasts$madForest2014 <- logElev
rasts

# Replace a layer:
rasts[[3]] <- forest2000
rasts

### GVectors

# example data
madDypsis <- fastData("madDypsis") # vector of points

# Convert SpatVector to GVector
dypsis <- fast(madDypsis)

### Retrieving GVector columns

dypsis$species # Returns the column

dypsis[[c("year", "species")]] # Returns a GRaster with these columns
dypsis[ , c("year", "species")] # Same as above

### Subsetting GVector geometries

# Subset first three geometries
dypsis[1:3]
dypsis[1:3, "species"]

# Get geometries by data table condition
dypsis[dypsis$species == "Dypsis betsimisarakae"]

### (Re)assigning GVector column values

# New column
dypsis$pi <- pi

# Re-assign values
dypsis$pi <- "pie"

# Re-assign specific values
dypsis$institutionCode[dypsis$institutionCode == "MO"] <-
   "Missouri Botanical Garden"

}
```

addCats,GRaster-method

Add rows or columns to the "levels" table of a categorical raster

Description

addCats() and addCats()<- add information to a categorical‘GRaster’s "levels" table.

- addCats()‘ uses data.table::merge() or cbind() to do this–it does not add new rows, but rather new columns.
- addCats()<- uses rbind() to add new categories (rows) to the "levels" table.

GRasters can represent categorical data (see vignette("GRasters", package = "fasterRaster")). Cell values are actually integers, each corresponding to a category, such as "desert" or "wetla (i.e., no categories–so not a categorical raster), or have at least two columns. The first column must have integers and represent raster values. One or more subsequent columns must have category labels. The column with these labels is the "active category".

Usage

```
## S4 method for signature 'GRaster'
addCats(x, value, merge = FALSE, layer = 1)

## S4 replacement method for signature 'GRaster'
addCats(x, layer = 1) <- value
```

Arguments

x	A GRaster.
value	A data.frame, data.table, a list of data.frames or data.tables with one per raster layer, or a categorical SpatRaster. The table's first column is the "value" column and must contain numeric values (of class numeric or character). If a SpatRaster is supplied, then its categories will be transferred to the GRaster.
merge	Logical (function addCats()): If FALSE (default), columns will be combined with the existing "levels" table using cbind(). If TRUE, they will be combined using data.table::merge().
layer	Numeric integers, logical vector, or character: Layer(s) to which to add or from which to drop levels.

Value

A GRaster. The "levels" table of the raster is modified.

See Also

terra::addCats(), concats(), combineLevels(), droplevels(), vignette("GRasters", package = "fasterRaster'

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data: Land cover raster
madCover <- fastData("madCover")

# Convert categorical SpatRaster to categorical GRaster:
cover <- fast(madCover)

### Properties of categorical rasters

cover # note categories
is.factor(cover) # Is the raster categorical?
nlevels(cover) # number of levels
levels(cover) # just the value and active column
cats(cover) # all columns
minmax(cover) # min/max values
minmax(cover, levels = TRUE) # min/max categories
catNames(cover) # column names of the levels table
missingCats(cover) # categories in table with no values in raster
freq(cover) # frequency of each category (number of cells)
zonalGeog(cover) # geometric statistics

### Active column

# Which column sets the category labels?
activeCat(cover)
activeCat(cover, names = TRUE)

activeCats(c(cover, cover))

# Choose a different column for category labels:
levels(cover)
activeCat(cover) <- 2
levels(cover)

### Managing levels tables

# Remove unused levels:
nlevels(cover)
cover <- droplevels(cover)
nlevels(cover)

# Re-assign levels:
value <- c(20, 30, 40, 50, 120, 130, 140, 170)
label <- c("Cropland", "Cropland", "Forest", "Forest",
 "Grassland", "Shrubland", "Herbaceous", "Flooded")

newCats <- data.frame(value = value, label = label)
```

```
cover <- categories(cover, layer = 1, value = newCats)
cats(cover)

# This is the same as:
levels(cover) <- newCats
cats(cover)

# Are there any values not assigned a category?
missingCats(cover)

# Let's assign a category for value 210 (water):
water <- data.frame(value = 210, label = "Water")
addCats(cover) <- water
levels(cover)

# Add more information to the levels table using merge():
landType <- data.frame(
     Value = c(20, 30, 40, 50, 120),
     Type = c("Irrigated", "Rainfed", "Broadleaf evergreen",
     "Broadleaf deciduous", "Mosaic with forest")
)
cats(cover)
cover <- addCats(cover, landType, merge = TRUE)
cats(cover)

### Logical operations on categorical rasters

cover < "Forest" # 1 for cells with a value < 40, 0 otherwise
cover <= "Forest" # 1 for cells with a value < 120, 0 otherwise
cover == "Forest" # 1 for cells with value of 40-120, 0 otherwise
cover != "Forest" # 1 for cells with value that is not 40-120, 0 otherwise
cover > "Forest" # 1 for cells with a value > 120, 0 otherwise
cover >= "Forest" # 1 for cells with a value >= 120, 0 otherwise

cover %in% c("Cropland", "Forest") # 1 for cropland/forest cells, 0 otherwise

### Combine categories from different rasters

# For the example, will create a second categorical raster fromm elevation.

# Divide elevation raster into "low/medium/high" levels:
madElev <- fastData("madElev")
elev <- fast(madElev)
elev <- project(elev, cover, method = "near") # convert to same CRS
fun <- "= if(madElev < 100, 0, if(madElev < 400, 1, 2))"
elevCat <- app(elev, fun)

levs <- data.frame(
     value = c(0, 1, 2),
     elevation = c("low", "medium", "high")
)
levels(elevCat) <- list(levs)
```

```
# Combine levels:
combined <- concats(cover, elevCat)
combined
levels(combined)

# Combine levels, treating value/NA combinations as new categories:
combinedNA <- concats(cover, elevCat, na.rm = FALSE)
combinedNA
levels(combinedNA)

}
```

---

addons *Test if addons directory exists and if an addon is installed*

Description

This function tests to see if the "addons" directory specified using `faster()` actually exists, and if a particular **GRASS** addons module is available. The addonsfolder and module must exists for methods that rely package = "fasterRaster")‘.

Usage

```
addons(x = NULL, fail = TRUE, verbose = TRUE)
```

Arguments

x	Either NULL or a character specifying the name of a **GRASS** addons module. If NULL, the existence of the addonsDir (see `faster()`) will be tested. If the module name is provided, the existence of the folder and module will be tested. The "/bin" subfolder should not be included.
fail	Logical: If TRUE (default), and the addons folder is not correctly specified, the exit the function with an error. If FALSE, then NULL will be returned with a warning.
verbose	Logical: If TRUE (default), display a message on success or warning (the fail option always displays a message).

Value

Logical.

See Also

```
vignette("addons", package = "fasterRaster")
```

Examples

```
if (grassStarted()) {

# Does the addons folder exist?
ao <- addons(fail = "warning")
if (ao) print("Addons is folder is probably correctly specified.")

# Does this particular module exist?
addon <- "v.centerpoint"
exten <- addons(addon, fail = FALSE)

if (exten) print("Extension `v.centerpoints` is installed.")

}
```

---

addTable<-,GVector,data.frame-method

Attach or detach GVector's data table

Description

`addTable()` adds an entire table to a `GVector`. It will replace any existing table. There must be one row in the table for each geometry (see `ngeom()`). You can also add a table column-by-column using the `$<-` operator.

`dropTable()` removes a data table associated with a `GVector`.

Usage

```
## S4 replacement method for signature 'GVector,data.frame'
addTable(x, ...) <- value

## S4 replacement method for signature 'GVector,data.table'
addTable(x, ...) <- value

## S4 replacement method for signature 'GVector,matrix'
addTable(x, ...) <- value

## S4 method for signature 'GVector'
dropTable(x)
```

Arguments

`x`	A `GVector`.
`...`	Other arguments (ignored).
`value`	A `data.frame`, `data.table`, or `matrix`.

Value

A GVector.

See Also

$<-, colbind(), rbind(), as.data.frame(), as.data.table()

Examples

```
if (grassStarted()) {

# Setup
library(sf)

# Rivers vector
madRivers <- fastData("madRivers")

# Convert sf to a GVector
rivers <- fast(madRivers)

# Convert GVector to data.frame or data.table
as.data.frame(rivers)
as.data.table(rivers)

# Subset rivers vector
rivers1 <- rivers[1:2]
rivers2 <- rivers[10:11]

# Concatenate rivers
riversCombo <- rbind(rivers1, rivers2)
riversCombo

# Add columns
newCol <- data.frame(new = 1:11)
riversCol <- colbind(rivers, newCol)
riversCol

# Remove table
riversCopy <- rivers
riversCopy # has data table
riversCopy <- dropTable(riversCopy)
riversCopy # no data table

# Add a new table
newTable <- data.frame(num = 1:11, letters = letters[1:11])
addTable(riversCopy) <- newTable
riversCopy

}
```

aggregate,GRaster-method

Aggregate raster cells into larger cells or combine geometries of a vector

Description

When applied to a GRaster, aggregate() creates a new raster with cells that are a multiple of the size of the cells of the original raster. The new cells can be larger or smaller than the original cells (this function thus emulates both the terra::aggregate() and terra::disagg() functions.)

When applied to a GVector, all geometries are combined into a "multipart" geometry, in which geometries are treated as if they were a single unit. Borders between aggregated geometries can be dissolved if the dissolve argument is TRUE. If the GVector has a data table associated with it, the output will also have a data table as long as there is at least one column with values that are all the same. Values of columns that do not have duplicated values will be converted to NA.

Usage

```
## S4 method for signature 'GRaster'
aggregate(
  x,
  fact = 2,
  fun = "mean",
  weight = FALSE,
  prob = NULL,
  na.rm = FALSE
)

## S4 method for signature 'GVector'
aggregate(x)
```

Arguments

x	A GRaster or GVector.
fact	Numeric vector (rasters only): One, two, or three positive values. These reflect the size of the new cells as multiples of the size of the old cells. If just one value is supplied, this is used for all two or three dimensions. If two values are supplied, the first is multiplied by the east-west size of cells, and the second north-south size of cells (the raster must be 2D). If three values are supplied, the third value is used as the multiplier of the vertical dimension of cells. Values are calculated using all cells that have their centers contained by the target cell. Note that unlike terra::aggregate() and terra::disagg(), these values need not be integers.
fun	Character (rasters only): Name of the function used to aggregate. For GRasters, this is the function that summarizes across cells. For GVectors, this function will be used to calculate new values of numeric or integer cells.

- mean: Average (default)
- median: Median
- mode: Most common value
- min: Minimum
- max: Maximum
- range: Difference between maximum and minimum
- sum: Sum
- varpop: Population variance
- sdpop: Population standard deviation
- quantile: Quantile (see argument prob)
- count: Number of non-NA cell
- diversity: Number of unique values

weight	Logical (rasters only): If FALSE, each source cell that has its center in the destination cell will be counted equally. If TRUE, the value of each source will be weighted the proportion of the destination cell the source cell covers.
prob	Numeric (rasters only): Quantile at which to calculate quantile.
na.rm	Logical (rasters only): If FALSE (default), propagate NA cells or NA values.

Value

A GRaster or GVector.

See Also

stats::aggregate(), terra::aggregate(), disagg(), terra::disagg()

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")
madCoast4 <- fastData("madCoast4")

### aggregating a GRaster

# Convert:
elev <- fast(madElev)

### Aggregate GRaster by same factor in 2 dimensions
# fasterRaster
agg2 <- aggregate(elev, 2, "mean")
agg2

# Compare rasters aggregated by fasterRaster and terra.
```

```
# These should be the same.
agg2terra <- aggregate(madElev, 2)

agg2 <- rast(agg2)
agg2 <- extend(agg2, agg2terra)
agg2 - agg2terra # value is ~0

### Aggregate GRaster by a non-integer factor in 2 dimensions
# fasterRaster
agg2.9 <- aggregate(elev, 2.9, "mean")
agg2.9

# terra
agg2.9terra <- aggregate(madElev, 2.9, "mean")
agg2.9terra

# Compare rasters aggregated by fasterRaster and terra.
# These should be different.
res(agg2.9)
res(agg2.9terra) # terra rounds aggregation factor down
2 * res(madElev) # original resolution multiplied by 2

### Aggregate GRaster by different factor in 2 dimensions
agg2x3 <- aggregate(elev, c(2, 3), "mean")
agg2x3

### aggregating a GVector

madCoast4 <- fastData("madCoast4")

# Convert:
coast4 <- fast(madCoast4)

# Aggregate and disaggregate:
aggCoast <- aggregate(coast4)
disaggCoast <- disagg(coast4)

ngeom(coast4)
ngeom(aggCoast)
ngeom(disaggCoast)

# plot
oldpar <- par(mfrow = c(1, 3))
plot(coast4, main = "Original", col = 1:nrow(coast4))
plot(aggCoast, main = "Aggregated", col = 1:nrow(aggCoast))
plot(disaggCoast, main = "Disaggregated", col = 1:nrow(disaggCoast))
par(oldpar)

}
```

---

app,GRaster-method *Apply a function to a set of rasters*

Description

app() applies a function to a set of "stacked" rasters. It is similar to the terra::app() and terra::lapp() functions.

appFuns() provides a table of **GRASS** functions that can be used by app() and their equivalents in **R**.

appCheck() tests whether a formula supplied to app() has any "forbidden" function calls.

The app() function operates in a manner somewhat different from terra::app(). The function to be applied *must* be written as a character string. For example, if the GRaster had layer names "x1" and "x2", then the function might be like "= max(sqrt(x1), log(x2))". Rasters **cannot** have the same names as functions used in the formula. In this example, the rasters could not be named "max", "sqrt", or "log". Note that the name of a GRaster is given by names()–this can be different from the name of the object in **R**.

The app() function will automatically check for GRaster names that appear also to be functions that appear in the formula. However, you can check a formula before running app() by using the appCheck() function. You can obtain a list of app() functions using appFuns(). Note that these are sometimes different from how they are applied in **R**.

Tips:

- Make sure your GRasters have names(). The function matches on these, not the name of the variable you use in **R** for the GRaster.
- Use null() instead of NA, and use isnull() instead of is.na().
- If you want to calculate values using while ignoring NA (or null) values, see the functions that begin with n (like nmean).
- Be mindful of the data type that a function returns. In **GRASS**, these are CELL (integer), FCELL (floating point values–precise to about the 7th decimal place), and DCELL (double-floating point values–precise to about the 15th decimal place; commensurate with the **R** numeric type). In cases where you want a GRaster to be treated like a float or double type raster, wrap the name of the GRaster in the float() or double() functions. This is especially useful if the GRaster might be assumed to be the CELL type because it only contains integer values. You can get the data type of a raster using datatype() with the type argument set to GRASS. You can change the data type of a GRaster using as.int(), as.float(), and as.doub(). Note that categorical rasters are really CELL (integer) rasters with an associated "levels" table. You can also change a CELL raster to a FCELL raster by adding then subtracting a decimal value, as in x - 0.1 + 0.1. See vignette("GRasters", package = "fasterRaster").
- The rand() function returns integer values by default. If you want non-integer values, use the tricks mentioned above to datatype non-integer values. For example, if you want uniform random values in the range between 0 and 1, use something like = float(rand(0 + 0.1, 1 + 0.1) - 0.1).

Usage

```
## S4 method for signature 'GRaster'
app(x, fun, datatype = "auto", seed = NULL)

appFuns(warn = TRUE)

## S4 method for signature 'GRaster,character'
appCheck(x, fun, msgOnGood = TRUE, failOnBad = TRUE)
```

Arguments

x	A GRaster with one or more named layers.
fun	Character: The function to apply. This must be written as a character string that follows these rules: • It must use typical arithmetic operators like +, -, *, / and/or functions that can be seen using appFuns(TRUE). • The names() of the rasters do not match any of the functions in the appFuns(TRUE) table. Note that x and y are forbidden names :(The help page for **GRASS** module r.mapcalc will be especially helpful. You can see this page using grassHelp("r.mapcalc").
datatype	Character: This ensures that rasters are treated as a certain type before they are operated on. This is useful when using rasters that have all integer values, which **GRASS** can assume represent integers, even if they are not supposed to. In this case, the output of operations on this raster might be an integer if otherwise not corrected. Partial matching is used, and options include: • "integer": Force all rasters to integers by truncating their values. The output may still be of type float if the operation creates non-integer values. • "float": Force rasters to be considered floating-point values. • "double": Force rasters to be considered double-floating point values. • "auto" (default): Ensure that rasters are represented by their native datatype() (i.e., "CELL" rasters as integers, "FCELL" rasters as floating-point, and "DCELL" as double-floating point).
seed	Numeric integer vector or NULL (default): A number for the random seed. Used only for app() function rand(), that generates a random number. If NULL, a seed will be generated. Defining the seed is useful for replicating a raster made with rand(). This must be an integer!
warn	Logical (function appFuns()): If TRUE (default), display a warning when allFuns() is not called interactively.
msgOnGood	Logical (function appCheck()): If TRUE (default), display a message if no overt problems with the raster names and formula are detected.
failOnBad	Logical (function appCheck()): If TRUE (default), fail if overt problems with raster names and the formula are detected.

Value

A GRaster.

See Also

terra::app(), terra::lapp(), subst(), classify(), and especially the **GRASS** manual page for module r.mapcalc (see grassHelp("r.mapcalc"))

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Elevation raster
madElev <- fastData("madElev")

# Convert SpatRaster to a GRaster:
elev <- fast(madElev)

# Create a "stack" of rasters for us to operate on:
x <- c(elev, elev^2, sqrt(elev))

# Demonstrate check for badly-named rasters:
names(x) <- c("cos", "asin", "exp")
fun <- "= cos / asin + exp"
appCheck(x, fun, failOnBad = FALSE)

# Rename rasters acceptable names and run the function:
names(x) <- c("x1", "x2", "x3")
fun <- "= (x1 / x2) + x3"
appCheck(x, fun, failOnBad = FALSE)
app(x, fun = fun)

# This is the same as:
(x[[1]] / x[[2]]) + x[[3]]

# We can view a table of app() functions using appFuns():
appFuns()

# We can also get the same table using:
data(appFunsTable)

# Apply other functions:
fun <- "= median(x1 / x2, x3, x1 * 2, cos(x2))"
app(x, fun = fun)

fun <- "= round(x1) * tan(x2) + log(x3, 10)"
app(x, fun = fun)

# Demonstrate effects of data type:
fun <- "= x1 + x3"
app(x, fun = fun, datatype = "float") # output is floating-point
app(x, fun = fun, datatype = "integer") # output is integer

# Some functions override the "datatype" argument:
fun <- "= sin(x2)"
app(x, fun = fun, datatype = "integer")

# Make a raster with random values [1:4], with equal probability of each:
```

```
fun <- "= round(rand(0.5, 4.5))"
rand <- app(elev, fun = fun)
rand

freqs <- freq(rand) # cell frequencies
print(freqs)

}
```

---

appFunsTable *Functions that can be used in app()*

Description

This is a table of functions that can be used in the `app()` function, their **R** equivalents, and the `datatype()` they return. You can view this table using ?appFunsTable or as a searchable, sortable **Shiny** table using `appFuns()`.

Format

A `data.frame`.

Source

OSGeo

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Elevation raster
madElev <- fastData("madElev")

# Convert SpatRaster to a GRaster:
elev <- fast(madElev)

# Create a "stack" of rasters for us to operate on:
x <- c(elev, elev^2, sqrt(elev))

# Demonstrate check for badly-named rasters:
names(x) <- c("cos", "asin", "exp")
fun <- "= cos / asin + exp"
appCheck(x, fun, failOnBad = FALSE)

# Rename rasters acceptable names and run the function:
names(x) <- c("x1", "x2", "x3")
fun <- "= (x1 / x2) + x3"
```

```
appCheck(x, fun, failOnBad = FALSE)
app(x, fun = fun)

# This is the same as:
(x[[1]] / x[[2]]) + x[[3]]

# We can view a table of app() functions using appFuns():
appFuns()

# We can also get the same table using:
data(appFunsTable)

# Apply other functions:
fun <- "= median(x1 / x2, x3, x1 * 2, cos(x2))"
app(x, fun = fun)

fun <- "= round(x1) * tan(x2) + log(x3, 10)"
app(x, fun = fun)

# Demonstrate effects of data type:
fun <- "= x1 + x3"
app(x, fun = fun, datatype = "float") # output is floating-point
app(x, fun = fun, datatype = "integer") # output is integer

# Some functions override the "datatype" argument:
fun <- "= sin(x2)"
app(x, fun = fun, datatype = "integer")

# Make a raster with random values [1:4], with equal probability of each:
fun <- "= round(rand(0.5, 4.5))"
rand <- app(elev, fun = fun)
rand

freqs <- freq(rand) # cell frequencies
print(freqs)

}
```

`Arith,GRaster,logical-method`

Arithmetic operations on GRasters

Description

`GRaster`**s**: You can do arithmetic operations on `GRaster`s and using normal operators in **R**: `+`, `-`, `*`, `/`, `^`, `%%` (modulus), and `%/%` (integer division).

`GVector`**s**: You can also do arithmetic operations on `GVector`s:

`+` operator: Same as `union()`

- operator: Same as erase()
* operator: Same as intersect()
/ operator: Same as xor()

Usage

```
## S4 method for signature 'GRaster,logical'
Arith(e1, e2)

## S4 method for signature 'logical,GRaster'
Arith(e1, e2)

## S4 method for signature 'GRaster,numeric'
Arith(e1, e2)

## S4 method for signature 'GRaster,integer'
Arith(e1, e2)

## S4 method for signature 'numeric,GRaster'
Arith(e1, e2)

## S4 method for signature 'integer,GRaster'
Arith(e1, e2)

## S4 method for signature 'GRaster,GRaster'
Arith(e1, e2)

## S4 method for signature 'GVector,GVector'
Arith(e1, e2)
```

Arguments

e1, e2	GRasters, numerics, integers, or logicals.

Value

A GRaster.

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Example data
madElev <- fastData("madElev")
```

```
# Convert a SpatRaster to a GRaster
elev <- fast(madElev)
elevs <- c(elev, elev, log10(elev) - 1, sqrt(elev))
names(elevs) <- c("elev1", "elev2", "log_elev", "sqrt_elev")

elev
elevs

# do some math
elev + 100
elev - 100
elev * 100
elev / 100
elev ^ 2
elev %/% 100 # divide then round down
elev %% 100 # modulus

100 + elev
100 %/% elev
100 %% elev

elevs + 100
100 + elevs

# math with logicals
elev + TRUE
elev - TRUE
elev * TRUE
elev / TRUE
elev ^ TRUE
elev %/% TRUE # divide then round down
elev %% TRUE # modulus

elevs + TRUE
TRUE + elevs

# Raster interacting with raster(s):
elev + elev
elev - elev
elev * elev
elev / elev
elev ^ log(elev)
elev %/% sqrt(elev) # divide then round down
elev %% sqrt(elev) # modulus

elevs + elev
elev * elevs

# sign
abs(-1 * elev)
abs(elevs)

# powers
```

```
sqrt(elevs)

# trigonometry
sin(elev)
cos(elev)
tan(elev)

asin(elev)
acos(elev)
atan(elev)

atan(elevs)
atan2(elev, elev^1.2)
atan2(elevs, elev^1.2)
atan2(elev, elevs^1.2)
atan2(elevs, elevs^1.2)

# logarithms
exp(elev)
log(elev)
ln(elev)
log2(elev)
log1p(elev)
log10(elev)
log10p(elev)
log(elev, 3)

log(elevs)

# rounding
round(elev + 0.5)
floor(elev + 0.5)
ceiling(elev + 0.5)
trunc(elev + 0.5)

}
```

`as.contour,GRaster-method`

Contour lines from a "GRaster"

Description

Create a `GVector` of contour lines from a `GRaster`.

Usage

```
## S4 method for signature 'GRaster'
as.contour(x, nlevels, levels)
```

Arguments

x	A GRaster.
nlevels	Numeric: A positive integer or missing (default). Number of levels at which to calculate contours. Levels will be calculated in equal-sized steps from the smallest to the largest value of x. Either nlevels or levels must be specified.
levels	Numeric vector: A numeric vector of values at which to calculate contour lines. Either nlevels or levels must be specified.

Value

A GVector representing contour lines.

See Also

terra::as.contour(), **GRASS** manual page for module r.contour (see grassHelp("r.contour"))

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Elevation raster
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster:
elev <- fast(madElev)

# Calculate contour lines:
conts <- as.contour(elev, nlevels = 10)

plot(elev)
plot(conts, add = TRUE)

}
```

---

as.data.frame,GVector-method

Convert GVector to a data frame

Description

Convert a GVector's data table to a data.frame or data.table.

Usage

```
## S4 method for signature 'GVector'
as.data.frame(x)

## S4 method for signature 'GVector'
as.data.table(x)
```

Arguments

x	A GVector.

Value

A data.frame or NULL (if the GRaster has no data table).

See Also

terra::as.data.frame(), data.table::as.data.table()

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Example data
madElev <- fastData("madElev")
madForest2000 <- fastData("madForest2000")
madCoast0 <- fastData("madCoast0")
madRivers <- fastData("madRivers")
madDypsis <- fastData("madDypsis")

### GRaster properties

# convert SpatRasters to GRasters
elev <- fast(madElev)
forest <- fast(madForest2000)

# plot
plot(elev)

dim(elev) # rows, columns, depths, layers
nrow(elev) # rows
ncol(elev) # columns
ndepth(elev) # depths
nlyr(elev) # layers

res(elev) # resolution
```

```
ncell(elev) # cells
ncell3d(elev) # cells (3D rasters only)

topology(elev) # number of dimensions
is.2d(elev) # is it 2D?
is.3d(elev) # is it 3D?

minmax(elev) # min/max values

# name of object in GRASS
sources(elev)

# "names" of the object
names(elev)

# coordinate reference system
crs(elev)

# extent (bounding box)
ext(elev)

# data type
datatype(elev)

# assigning
copy <- elev
copy[] <- pi # assign all cells to the value of pi
copy

# concatenating multiple GRasters
rasts <- c(elev, forest)
rasts

# adding a raster "in place"
add(rasts) <- ln(elev)
rasts

# subsetting
rasts[[1]]
rasts[["madForest2000"]]

# assigning
rasts[[4]] <- elev > 500

# number of layers
nlyr(rasts)

# names
names(rasts)
names(rasts) <- c("elev_meters", "forest", "ln_elev", "high_elevation")
rasts

### GVector properties
```

```
# convert sf vectors to GVectors
coast <- fast(madCoast4)
rivers <- fast(madRivers)
dypsis <- fast(madDypsis)

# extent
ext(rivers)

W(rivers) # western extent
E(rivers) # eastern extent
S(rivers) # southern extent
N(rivers) # northern extent
top(rivers) # top extent (NA for 2D rasters like this one)
bottom(rivers) # bottom extent (NA for 2D rasters like this one)

# coordinate reference system
crs(rivers)
st_crs(rivers)

# column names and data types
names(coast)
datatype(coast)

# name of object in GRASS
sources(rivers)

# points, lines, or polygons?
geomtype(dypsis)
geomtype(rivers)
geomtype(coast)

is.points(dypsis)
is.points(coast)

is.lines(rivers)
is.lines(dypsis)

is.polygons(coast)
is.polygons(dypsis)

# dimensions
nrow(rivers) # how many spatial features
ncol(rivers) # hay many columns in the data frame

# number of geometries and sub-geometries
ngeom(coast)
nsubgeom(coast)

# 2- or 3D
topology(rivers) # dimensionality
is.2d(elev) # is it 2D?
is.3d(elev) # is it 3D?
```

```
# Update values from GRASS
# (Reads values from GRASS... will not appear to do anything in this case)
coast <- update(coast)

### operations on GVectors

# convert to data frame
as.data.frame(rivers)
as.data.table(rivers)

# subsetting
rivers[c(1:2, 5)] # select 3 rows/geometries
rivers[-5:-11] # remove rows/geometries 5 through 11
rivers[ , 1] # column 1
rivers[ , "NAM"] # select column
rivers[["NAM"]] # select column
rivers[1, 2:3] # row/geometry 1 and column 2 and 3
rivers[c(TRUE, FALSE)] # select every other geometry (T/F vector is recycled)
rivers[ , c(TRUE, FALSE)] # select every other column (T/F vector is recycled)

# removing data table
noTable <- dropTable(rivers)
noTable
nrow(rivers)
nrow(noTable)

# Refresh values from GRASS
# (Reads values from GRASS... will not appear to do anything in this case
# since the rivers object is up-to-date):
rivers <- update(rivers)

# Concatenating multiple vectors
rivers2 <- rbind(rivers, rivers)
dim(rivers)
dim(rivers2)

}
```

`as.int,GRaster-method` *Coerce raster to integer, float, or double precision*

Description

In **fasterRaster**, rasters can have three data types: "factor" (categorical rasters), "integer" (integers), "float" (floating point values, accurate to 6th to 9th decimal places), and "double" (double-precision values, accurate to the 15th to 17th decimal places). The type of raster can be checked with:

- `as.int()`: Coerce values to integers (**GRASS** type `CELL`).
- `as.float()`: Coerce values to floating-point precision.

- as.doub(): Coerce values to double-floating point precision.
- Integer rasters can be converted categorical rasters by adding "levels" tables with levels<- or categories().

Usage

```
## S4 method for signature 'GRaster'
as.int(x)

## S4 method for signature 'GRaster'
as.float(x)

## S4 method for signature 'GRaster'
as.doub(x)
```

Arguments

x	A GRaster.

Value

A GRaster.

See Also

datatype(), terra::datatype(), is.int(), is.float(), is.doub(), levels<-, vignette("GRasters", package = "fasterRaster")

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")
madForest2000 <- fastData("madForest2000")

# Convert SpatRasters to GRasters
elev <- fast(madElev)
forest <- fast(madForest2000)

### GRaster properties

# plotting
plot(elev)

# dimensions
dim(elev) # rows, columns, depths, layers
nrow(elev) # rows
```

```
ncol(elev) # columns
ndepth(elev) # depths
nlyr(elev) # layers

res(elev) # resolution (2D)
res3d(elev) # resolution (3D)
zres(elev) # vertical resolution
xres(elev) # vertical resolution
yres(elev) # vertical resolution
zres(elev) # vertical resolution (NA because this is a 2D GRaster)

# cell counts
ncell(elev) # cells
ncell3d(elev) # cells (3D rasters only)

# number of NA and non-NA cells
nacell(elev)
nonnacell(elev)

# topology
topology(elev) # number of dimensions
is.2d(elev) # is it 2-dimensional?
is.3d(elev) # is it 3-dimensional?

minmax(elev) # min/max values

# "names" of the object
names(elev)

# coordinate reference system
crs(elev)
st_crs(elev)
coordRef(elev)

# extent (bounding box)
ext(elev)

# vertical extent (not defined for this raster)
zext(elev)

# data type
datatype(elev) # fasterRaster type
datatype(elev, "GRASS") # GRASS type
datatype(elev, "terra") # terra type
datatype(elev, "GDAL") # GDAL type

is.integer(elev)
is.float(elev)
is.double(elev)
is.factor(elev)

# convert data type
as.int(elev) # integer; note that "elev" is already of type "integer"
```

```
as.float(elev) # floating-precision
as.doub(elev) # double-precision

# assigning
pie <- elev
pie[] <- pi # assign all cells to the value of pi
pie

# concatenating multiple GRasters
rasts <- c(elev, forest)
rasts

# subsetting
rasts[[1]]
rasts[["madForest2000"]]

# replacing
rasts[[2]] <- 2 * forest
rasts

# adding layers
rasts[[3]] <- elev > 500 # add a layer
rasts <- c(rasts, sqrt(elev)) # add another
add(rasts) <- ln(elev)
rasts

# names
names(rasts)
names(rasts) <- c("elev_meters", "2_x_forest", "high_elevation", "sqrt_elev", "ln_elev")
rasts

# remove a layer
rasts[["2_x_forest"]] <- NULL
rasts

# number of layers
nlyr(rasts)

# correlation and covariance matrices
madLANDSAT <- fastData("madLANDSAT")
landsat <- fast(madLANDSAT) # projects matrix
layerCor(landsat) # correlation
layerCor(landsat, fun = 'cov') # covariance

}
```

---

as.lines,GRaster-method

Convert a raster to a lines vector

Description

as.lines() converts a GRaster to a "lines" GVector. Before you apply this function, you may need to run thinLines() on the raster to reduce linear features to a single-cell width. You may also need to use clean geometry (especially the removeDups() and removeDangles()) afterward to remove duplicated vertices and "dangling" lines.

Usage

```
## S4 method for signature 'GRaster'
as.lines(x)
```

Arguments

x	A GRaster. If more than one layer is in the GRaster, only the first will be used (with a warning).

Value

A GVector.

See Also

as.points(), as.polygons(), terra::as.lines(), thinLines(), geometry cleaning, and **GRASS** module r.to.vect

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Elevation
madElev <- fastData("madElev")

# Convert to GRaster:
elev <- fast(madElev)

# Thin elevation raster:
thinned <- thinLines(elev, iter = 300)
plot(thinned)

# Convert to lines:
rastToLines <- as.lines(thinned)
plot(rastToLines)

# We can clean this:
cleanLines <- fixDangles(x = rastToLines)
plot(rastToLines, col = "red")
plot(cleanLines, add = TRUE)
```

```
}
```

`as.points,GRaster-method`
Convert a GRaster, or lines or polygons GVector to a points vector

---

Description

`as.points()` converts a `GRaster`, or a lines or polygons `GVector` to a points `GVector`.

For `GRasters`, the points have the coordinates of cell centers and are assigned the cells' values. Only non-`NA` cells will be converted to points.

For `GVectors`, each point will have the attributes of the line or polygon to which it belonged. Points are extracted from each vertex.

Usage

```
## S4 method for signature 'GRaster'
as.points(x, values = TRUE)

## S4 method for signature 'GVector'
as.points(x)
```

Arguments

`x`	A `GRaster`, `GVector`.
`values`	Logical: If `TRUE` (default), create an attribute table with raster cell values, with one row per point.

Value

A `points` `GVector`.

See Also

`crds()`, `as.lines()`, `as.polygons()`, `terra::as.points()`, and modules `v.to.points` and `r.to.vect` in **GRASS**

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Elevation raster, outline of a part of Madagascar, and rivers vector:
madElev <- fastData("madElev")
madCoast0 <- fastData("madCoast0")
```

```
madRivers <- fastData("madRivers")

# Convert to GRaster and GVectors:
elev <- fast(madElev)
coast <- fast(madCoast0)
rivers <- fast(madRivers)

# For this example, we will first crop to a small extent.
river <- rivers[1]
elevCrop <- crop(elev, river)
elevPoints <- as.points(elevCrop)
elevPoints

plot(elevCrop)
plot(elevPoints, pch = '.', add = TRUE)

# Extract points from vectors:
coastPoints <- as.points(coast)
riversPoints <- as.points(rivers)

plot(coast)
plot(coastPoints, add = TRUE)

plot(rivers, col = "blue", add = TRUE)
plot(riversPoints, col = "blue", add = TRUE)

}
```

`as.polygons,GRaster-method`

Convert a raster to a polygons vector

Description

`as.polygons()` converts a `GRaster` to a "polygons" `GVector`. After running this function, geometry cleaning may be useful to use to "tidy up" the vector.

Usage

```
## S4 method for signature 'GRaster'
as.polygons(x, round = TRUE, smooth = FALSE)
```

Arguments

`x`	A `GRaster`. If more than one layer is in the `GRaster`, only the first will be used (with a warning).
`round`	Logical: If `TRUE` (default), values in the raster will be rounded first before conversion to a vector. This causes cells that are adjacent that have the same (rounded) values to be combined into a single polygon. For more control, see `clump()`.

smooth Logical: If TRUE, round the corners of square features. Default is FALSE.

Value

A GVector.

See Also

as.points(), as.lines(), terra::as.polygons(), geometry cleaning, and **GRASS** module r.to.vect

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Elevation
madElev <- fastData("madElev")

# Convert SpatRaster to GRaster:
elev <- fast(madElev)

# To speed things up, first group cells of similar value:
elevClumps <- clump(elev, minDiff = 0.0115)

# Convert to polygons:
rastToPolys <- as.polygons(elevClumps)
plot(rastToPolys)

}
```

---

bioclims,GRaster-method

BIOCLIM rasters

Description

The BIOCLIM set of bioclimatic variables were created for modeling species' geographic distributions (Booth et al. 2014). This function can create the "standard" 19 set of variables, plus several more from an "extended" set.

"Classic" set of BIOCLIM variables (Booth et al. 2014): The units reported below assume that input rasters are in mm (precipitation) and deg C (temperature), and that each raster represents a month (but other time units are allowed, with corresponding changes to the temporal units assumed below).

- BIO1: Mean annual temperature, calculated using monthly means (deg C)

- BIO2: Mean diurnal range across months (average of monthly difference between maximum and minimum temperature) (deg C)
- BIO3: Isothermality (100 * BIO02 / BIO07; unit-less)
- BIO4: Temperature seasonality (standard deviation across months of average monthly temperature * 100; deg C)
- BIO5: Maximum temperature of the warmest month (based on maximum temperature; deg C)
- BIO6: Minimum temperature of the coldest month (based on minimum temperature; deg C)
- BIO7: Range of annual temperature (BIO05 - BIO06; deg C)
- BIO8: Temperature of the wettest quarter (based on mean temperature; deg C)
- BIO9: Temperature of the driest quarter (based on mean temperature; deg C)
- BIO10: Temperature of the warmest quarter (based on mean temperature; deg C)
- BIO11: Temperature of the coldest quarter (based on mean temperature; deg C)
- BIO12: Total annual precipitation (mm)
- BIO13: Precipitation of the wettest month (mm)
- BIO14: Precipitation of the driest month (mm)
- BIO15: Precipitation seasonality (100 * coefficient of variation; unit-less)
- BIO16: Precipitation of the wettest quarter (mm)
- BIO17: Precipitation of the driest quarter (mm)
- BIO18: Precipitation of the warmest quarter (based on mean temperature; mm)
- BIO19: Precipitation of the coldest quarter (based on mean temperature; mm)

"Extended" set of BIOCLIM variables (starts at 41 to avoid conflicts with Kriticos et al. 2014):

- BIO41: Temperature of the quarter following the coldest quarter (based on mean temperature; deg C)
- BIO42: Temperature of the quarter following the warmest quarter (based on mean temperature; deg C)
- BIO43: Precipitation of the quarter following the coldest quarter (based on mean temperature; mm)
- BIO44: Precipitation of the quarter following the warmest quarter (based on mean temperature; mm)
- BIO45: Temperature of the quarter following the driest quarter (based on mean temperature; deg C)
- BIO46: Temperature of the quarter following the wettest quarter (based on mean temperature; deg C)
- BIO47: Precipitation of the quarter following the driest quarter (based on mean temperature; mm)
- BIO48: Precipitation of the quarter following the wettest quarter (based on mean temperature; mm)
- BIO49: Hottest month (based on maximum temperature)
- BIO50: Coldest month (based on minimum temperature)

- BIO51: Wettest month
- BIO52: Driest month
- BIO53: First month of the warmest quarter (based on mean temperature)
- BIO54: First month of the coldest quarter (based on mean temperature)
- BIO55: First month of the wettest quarter
- BIO56: First month of the driest quarter
- BIO57: The greatest decrease in temperature from one month to the next (deg C; always >= 0)
- BIO58: The greatest increase in temperature from one month to the next (deg C; always >= 0)
- BIO59: The greatest decrease in precipitation from one month to the next (mm; always >= 0)
- BIO60: The greatest increase in precipitation from one month to the next (mm; always >= 0)

By default, "quarter" refers to any consecutive run of three months, not a financial quarter. A quarter can thus include November-December-January, or December-January-February, for example. However, the length of a quarter can be changed using the argument `quarter`.

The variables are defined assuming that the input rasters represent monthly values (12 rasters for min/max temperature and precipitation), but you can also use sets of 52 rasters, representing one per week, in which case "quarter" would be a successive run of 3 weeks. You could also attempt 365 rasters, in which case a "quarter" would be a run of 3 successive days.

BIOCLIMs 41 through 44 are added here to capture the "shoulder" seasons (spring and autumn) important in temperature regions. BIOCLIMs 45 through 48 are also included for consistency.

BIOCLIMs 49 through 60 are not bioclimatic variables per se, but useful for assessing the properties of the variables that are defined based on the "-est" month or quarter.

Usage

```
## S4 method for signature 'GRaster'
bioclims(
  ppt,
  tmin,
  tmax,
  tmean = NULL,
  bios = NULL,
  sample = TRUE,
  quarter = 3,
  pptDelta = 1,
  verbose = TRUE
)

## S4 method for signature 'SpatRaster'
bioclims(
  ppt,
  tmin,
  tmax,
  tmean = NULL,
```

```
  bios = NULL,
  sample = TRUE,
  quarter = 3,
  pptDelta = 1,
  verbose = TRUE
)
```

Arguments

ppt	A multi-layered GRaster or SpatRaster, representing monthly/weekly/daily precipitation.
tmin, tmax	A multi-layered GRaster or SpatRaster, representing monthly/weekly/daily minimum and maximum temperature.
tmean	Either NULL (default), or a multi-layered GRaster or SpatRaster, representing monthly/weekly/daily average temperature. If NULL, tmean will be calculated internally from tmin and tmax. Providing these rasters thus saves time if you already have them on hand.
bios	Any of: • Numeric values: Calculate these BIOCLIM variables. For example, bios = c(1, 12) calculates BIOCLIMs 1 and 12. • NULL (default): Calculate BIOCLIMs 1 through 19 • "*": Calculate all BIOCLIMs this function can calculate. • "+": Calculate BIOCLIMs 41 onward. • Any combination of the above except NULL (e.g., c(1, 12, "+")).
sample	Logical: If TRUE (default), BIO4 and 15 are calculated with the sample standard deviation. If FALSE, then the population standard deviation is used.
quarter	Numeric: Length of a "quarter". BIOCLIM variables are typically calculated using monthly-averaged rasters (e.g., precipitation and temperature of January, February, etc.), in which case a "quarter" is 3 months (so the default for quarter is 3). However, this function can accommodate any set of rasters representing a time series (e.g., 365 for daily rasters), in which case the user can decide what constitutes a "quarter" for calculation of the any BIOCLIMs that use "quarters" in their definitions.
pptDelta	Numeric: Value to add to precipitation for calculation of BIO15 (coefficient of variation of precipitation, times 100). Adding a small value avoids division by 0. The default is 1.
verbose	Logical: If TRUE (default), display progress.

Value

A GRaster with one or more layers.

References

Booth, T.H., Nix, H.A., Busby, J.R., and Hutchinson, M.F. 2014. BIOCLIM: The first species distribution modeling package, its early applications and relevance to most current MaxEnt studies. *Diversity and Distributions* 20:1-9 doi:10.1111/ddi.12144.

Kriticos, D.J., Jarošik, V., and Otam N. 2014. Extending the suite of BIOCLIM variables: A proposed registry system and case study using principal components analysis. *Methods in Ecology and Evolution* 5:956-960 doi:10.1111/2041210X.12244.

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Load rasters with precipitation and min/max temperature
madPpt <- fastData("madPpt")
madTmin <- fastData("madTmin")
madTmax <- fastData("madTmax")

### Classic and extended BIOCLIMs from SpatRasters
bcSR <- bioclims(madPpt, madTmin, madTmax, bios = "*")
bcSR

### BIOCLIMs from GRasters
ppt <- fast(madPpt)
tmin <- fast(madTmin)
tmax <- fast(madTmax)

# For small rasters, takes longer to run compared to SpatRaster version:
bc <- bioclims(ppt, tmin, tmax, bios = c(1, 5, 12))
bc
plot(bc)

}
```

---

breakPolys,GVector-method

Fix issues with geometries of a vector

Description

These functions are intended to help fix geometric issues with a `GVector`. Note that the functionality of the `snap()` and `removeAreas()` functions can also be implemented when using `fast()` to create a `GVector`.

- `breakPolys()`: Break topologically clean areas. This is similar to `fixLines()`, except that it does not break loops. Topologically clean vectors may occur if the vector was imported from a format that does not enforce topology, such as a shapefile. Duplicate geometries are automatically removed after breaking.
- `fixBridges()`: Change "bridges" to "islands" (which are topologically incorrect) within geometries to lines.

- `fixDangles()`: Change "dangles" hanging off boundaries to lines if shorter than `tolerance` distance. If `tolerance` is <0, all dangles will be changed to lines. Units of `tolerance` are in map units, or in degrees for unprojected CRSs. If `tolerance` <0, all dangles are removed, and the function will retain only closed loops and lines connecting loops. Dangles will be removed from longest to shortest.
- `fixLines()`: Break lines at intersections and lines that form closed loops.
- `remove0()`: Remove all boundaries and lines with a length of 0.
- `removeAngles()`: Collapse lines that diverge at an angle that is computationally equivalent to 0. This tool often needs to be followed with the `break()` and `removeDups()` methods.
- `removeBridges()`: Remove "bridges" to "islands" (which are topologically incorrect) within geometries.
- `removeDangles()`: Remove "dangling" lines if shorter than `tolerance` distance. If `tolerance` is <0, all dangles will be removed. Units of `tolerance` are in map units, or in degrees for unprojected CRSs. If `tolerance` <0, all dangles are removed, and the function will retain only closed loops and lines connecting loops. Dangles will be removed from longest to shortest.
- `removeDupCentroids()`: Remove duplicated area centroids. In **GRASS**, closed polygons have their attributes mapped to a (hidden) centroid of the polygon.
- `removeDups()`: Remove duplicated features and area centroids.
- `removeSmallPolys()`: Remove polygons smaller than `tolerance`. Units of `tolerance` are in square meters (regardless of the CRS).
- `snap()`: Snap lines/boundaries to each other if they are less than `tolerance` apart. Subsequent removal of dangles may be needed. Units of `tolerance` are map units, or degrees for unprojected CRSs.

Usage

```
## S4 method for signature 'GVector'
breakPolys(x)

## S4 method for signature 'GVector'
fixBridges(x)

## S4 method for signature 'GVector'
fixDangles(x, tolerance = -1)

## S4 method for signature 'GVector'
fixLines(x)

## S4 method for signature 'GVector'
remove0(x)

## S4 method for signature 'GVector'
removeAngles(x)

## S4 method for signature 'GVector'
removeBridges(x)
```

```
## S4 method for signature 'GVector'
removeDangles(x, tolerance = -1)

## S4 method for signature 'GVector'
removeDupCentroids(x)

## S4 method for signature 'GVector'
removeDups(x)

## S4 method for signature 'GVector'
removeSmallPolys(x, tolerance)

## S4 method for signature 'GVector'
snap(x, tolerance)
```

Arguments

x	A GVector.
tolerance	Numeric or NULL (default): Minimum distance in map units (degrees for unprojected, usually meters for projected) or minimum area (in meters-squared, regardless of projection).

Value

A GVector.

See Also

terra::topology(), fillHoles(), terra::removeDupNodes(), *Details* section in fast(), simplifyGeom(), smoothGeom(), **GRASS** manual page for module v.clean (see grassHelp("v.clean"))

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Example data
madRivers <- fastData("madRivers")
rivers <- fast(madRivers)
soam <- rivers[rivers$NAM == "SOAMIANINA"] # select one river for illustration

### Simplify geometry (remove nodes)

vr <- simplifyGeom(soam, tolerance = 2000)
dp <- simplifyGeom(soam, tolerance = 2000, method = "dp")
dpr <- simplifyGeom(soam, tolerance = 2000, method = "dpr", prop = 0.5)
```

```
rw <- simplifyGeom(soam, tolerance = 2000, method = "rw")

plot(soam, col = "black", lwd = 3)
plot(vr, col = "blue", add = TRUE)
plot(dp, col = "red", add = TRUE)
plot(dpr, col = "chartreuse", add = TRUE)
plot(rw, col = "orange", add = TRUE)

legend("bottom",
   xpd = NA,
   legend = c(
   "Original",
      "Vertex reduction",
      "Douglas-Peucker",
      "Douglas-Peucker reduction",
      "Reumann-Witkam"
 ),
 col = c("black", "blue", "red", "chartreuse", "orange"),
 lwd = c(3, 1, 1, 1, 1)
)

### Smooth geometry

hermite <- smoothGeom(soam, dist = 2000, angle = 3)
chaiken <- smoothGeom(soam, method = "Chaiken", dist = 2000)

plot(soam, col = "black", lwd = 2)
plot(hermite, col = "blue", add = TRUE)
plot(chaiken, col = "red", add = TRUE)

legend("bottom",
   xpd = NA,
   legend = c(
   "Original",
      "Hermite",
      "Chaiken"
 ),
 col = c("black", "blue", "red"),
 lwd = c(2, 1, 1, 1, 1)
)

### Clean geometry

# Has no effect on this vector!
noDangs <- removeDangles(soam, tolerance = 10000)

plot(soam, col = "black", lwd = 2)
plot(noDangs, col = "red", add = TRUE)

legend("bottom",
   xpd = NA,
   legend = c(
   "Original",
```

```
      "No dangles"
 ),
 lwd = c(2, 1),
 col = c("black", "red")
)

}
```

`buffer,GRaster-method` *Increase/decrease the size of a vector or around non-NA cells of a raster*

Description

Buffers can be constructed for `GRasters` or `GVectors`. For rasters, the `buffer()` function creates a buffer around non-`NA` cells. The output will be a raster. For vectors, the `buffer()` and `st_buffer()` functions create a vector polygon larger or smaller than the focal vector.

Usage

```
## S4 method for signature 'GRaster'
buffer(
  x,
  width,
  unit = "meters",
  method = "Euclidean",
  background = 0,
  lowMemory = FALSE
)

## S4 method for signature 'GVector'
buffer(x, width, capstyle = "round", dissolve = TRUE)

## S4 method for signature 'GVector'
st_buffer(x, dist, endCapStyle = "round", dissolve = FALSE)
```

Arguments

`x`	A `GRaster` or `GVector`.
`width`	Numeric: For rasters – Maximum distance cells must be from focal cells to be within the buffer. For rasters, if the buffering unit is `"cells"`, then to get `n` cell widths, use `n + epsilon`, where `epsilon` is a small number (e.g., 0.001). The larger the buffer, this smaller this must be to ensure just `n` cells are included. For vectors, distance from the object to place the buffer. Negative values create "inside" buffers. Units are in the same units as the current coordinate reference system (e.g., degrees for WGS84 or NAD83, often meters for projected systems).

`unit` Character: Rasters – Indicates the units of `width`. Can be one of:

`* `"cells"`: Units are numbers of cells.`

- `"meters"` (default), `"metres"`, or `"m"`; `"kilometers"` or `"km"`; `"feet"` or `"ft"`; `"miles"` or `"mi"`; `"nautical miles"` or `"nmi"`. Partial matching is used and case is ignored.

`method` Character: Rasters – Only used if `units` is `"cells"`. Indicates the manner in which distances are calculated for adding of cells:

- `"Euclidean"`: Euclidean distance (default)
- `"Manhattan"`: "taxi-cab" distance
- `"maximum"`: Maximum of the north-south and east-west distances between points.

Partial matching is used and case is ignored.

`background` Numeric: Rasters – Value to assign to cells that are not `NA` and not part of the buffer (default is 0).

`lowMemory` Logical: Rasters – Only used if buffering a raster and `units` is not `"meters"`. If `FALSE` (default) use faster, memory-intensive procedure. If `TRUE` then use the slower, low-memory version. To help decide which to use, consider using the low-memory version on a system with 1 GB of RAM for a raster larger than about 32000x32000 cells, or for a system with with 8 GB of RAM a raster larger than about 90000x90000 cells.

`capstyle`, `endCapStyle`
Character: Vectors – Style for ending the "cap" of buffers around lines. Valid options include `"rounded"`, `"square"`, and `"flat"`.

`dissolve` Logical (`GVectors`): If `TRUE` (default), dissolve all buffers after creation. If `FALSE`, construct a buffer for each geometry. Note that overlapping buffers can cause this function to fail because it creates a topologically ambiguous polygon. Thus, using `dissolve = TRUE` is recommended.

`dist` Vectors – Same as `width`.

Details

Note that in some cases, topologically incorrect vectors can be created when buffering. This can arise when buffers intersect to create intersections that technically belong to two or more geometries. This issue can be resolved by dissolving borders between buffered geometries using `dissolve = TRUE`, but as of now, there is no fix if you do not want to dissolve geometries. A workaround would be to create a different `GVector` for each geometry, and then buffer each individually :(.

Value

A `GRaster` or a `GVector`.

See Also

`terra::buffer()`, `sf::st_buffer()`

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Elevation raster, rivers vector
madElev <- fastData("madElev")
madRivers <- fastData("madRivers")

# Convert a SpatRaster to a GRaster, and sf to a GVector
elev <- fast(madElev)
rivers <- fast(madRivers)

### Buffer a raster by a given distance:
buffByDist <- buffer(elev, width = 2000) # 2000-m buffer
plot(buffByDist, legend = FALSE)
plot(madElev, add = TRUE)

### Buffer a raster by a given number of cells:
buffByCells <- buffer(elev, width = 20.01, unit = "cells") # 20-cell buffer
plot(buffByCells, legend = FALSE)
plot(madElev, add = TRUE)

### Buffer a vector:
buffRivers <- buffer(rivers, width = 2000, dissolve = TRUE) # 2000-m buffer
plot(buffRivers)
plot(st_geometry(madRivers), col = "blue", add = TRUE)

}
```

---

c,GRaster-method *"Stack" GRasters*

Description

GRasters can be "stacked" using this function, effectively creating a multi-layered raster. This is different from creating a 3-dimensional raster, though such an effect can be emulated using stacking. GVectors can be combined into a single vector. Stacks can only be created when:

- All objects are the same class (either all GRasters or all GVectors).
- All objects have the same coordinate reference system (see crs()).
- Horizontal extents are the same (see ext()).
- Horizontal dimensions are the same (see res()).
- The topology (2- or 3-dimensional) must be the same. If 3D, then all rasters must have the same number of depths and vertical extents (see topology()).

Data tables associated with GVectors will be combined if each vector has a table and if each table has the same columns and data types. Otherwise, the data table will be combined using merge().

Usage

```
## S4 method for signature 'GRaster'
c(x, ...)
```

Arguments

x	A GRaster or a GVector.
...	One or more GRasters, one or more GVectors, a list of GRasters, or a list of GVectors. You can use a mix of lists and individual rasters or vectors.

Value

A GRaster.

See Also

add<-, terra::c(), add<-

Examples

```
if (grassStarted()) {

# Setup
madForest2000 <- fastData("madForest2000")
madForest2014 <- fastData("madForest2014")

# Convert SpatRasters to GRasters:
forest2000 <- fast(madForest2000)
forest2014 <- fast(madForest2014)

# Combine:
forest <- c(forest2000, forest2014)
forest

nlyr(forest)

}
```

---

catNames,GRaster-method

Names of columns of the levels table of a categorical raster

Description

This function returns the column names of each "levels" table of a categorical raster (see vignette("GRasters", package = "fasterRaster")).

Usage

```
## S4 method for signature 'GRaster'
catNames(x, layer = NULL)

## S4 method for signature 'SpatRaster'
catNames(x, layer = NULL)
```

Arguments

x	A GRaster.
layer	NULL, numeric integer, or character: The index (indices) or name(s) of one or more raster layers. The default is NULL, in which case all names for all layers are returned.

Value

A list of character vectors.

See Also

cats(), vignette("GRasters", package = "fasterRaster")

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data: Land cover raster
madCover <- fastData("madCover")

# Convert categorical SpatRaster to categorical GRaster:
cover <- fast(madCover)

### Properties of categorical rasters

cover # note categories
is.factor(cover) # Is the raster categorical?
nlevels(cover) # number of levels
levels(cover) # just the value and active column
cats(cover) # all columns
minmax(cover) # min/max values
minmax(cover, levels = TRUE) # min/max categories
catNames(cover) # column names of the levels table
missingCats(cover) # categories in table with no values in raster
freq(cover) # frequency of each category (number of cells)
zonalGeog(cover) # geometric statistics

### Active column
```

```
# Which column sets the category labels?
activeCat(cover)
activeCat(cover, names = TRUE)

activeCats(c(cover, cover))

# Choose a different column for category labels:
levels(cover)
activeCat(cover) <- 2
levels(cover)

### Managing levels tables

# Remove unused levels:
nlevels(cover)
cover <- droplevels(cover)
nlevels(cover)

# Re-assign levels:
value <- c(20, 30, 40, 50, 120, 130, 140, 170)
label <- c("Cropland", "Cropland", "Forest", "Forest",
 "Grassland", "Shrubland", "Herbaceous", "Flooded")

newCats <- data.frame(value = value, label = label)

cover <- categories(cover, layer = 1, value = newCats)
cats(cover)

# This is the same as:
levels(cover) <- newCats
cats(cover)

# Are there any values not assigned a category?
missingCats(cover)

# Let's assign a category for value 210 (water):
water <- data.frame(value = 210, label = "Water")
addCats(cover) <- water
levels(cover)

# Add more information to the levels table using merge():
landType <- data.frame(
     Value = c(20, 30, 40, 50, 120),
     Type = c("Irrigated", "Rainfed", "Broadleaf evergreen",
     "Broadleaf deciduous", "Mosaic with forest")
)
cats(cover)
cover <- addCats(cover, landType, merge = TRUE)
cats(cover)

### Logical operations on categorical rasters

cover < "Forest" # 1 for cells with a value < 40, 0 otherwise
```

```
cover <= "Forest" # 1 for cells with a value < 120, 0 otherwise
cover == "Forest" # 1 for cells with value of 40-120, 0 otherwise
cover != "Forest" # 1 for cells with value that is not 40-120, 0 otherwise
cover > "Forest" # 1 for cells with a value > 120, 0 otherwise
cover >= "Forest" # 1 for cells with a value >= 120, 0 otherwise

cover %in% c("Cropland", "Forest") # 1 for cropland/forest cells, 0 otherwise

### Combine categories from different rasters

# For the example, will create a second categorical raster fromm elevation.

# Divide elevation raster into "low/medium/high" levels:
madElev <- fastData("madElev")
elev <- fast(madElev)
elev <- project(elev, cover, method = "near") # convert to same CRS
fun <- "= if(madElev < 100, 0, if(madElev < 400, 1, 2))"
elevCat <- app(elev, fun)

levs <- data.frame(
     value = c(0, 1, 2),
     elevation = c("low", "medium", "high")
)
levels(elevCat) <- list(levs)

# Combine levels:
combined <- concats(cover, elevCat)
combined
levels(combined)

# Combine levels, treating value/NA combinations as new categories:
combinedNA <- concats(cover, elevCat, na.rm = FALSE)
combinedNA
levels(combinedNA)

}
```

---

cellSize,GRaster-method

Area of GRaster cells

Description

`cellArea()` returns a raster will cell values equal to their area. To get the area of all cells of a raster, see `expanse()`.

Usage

```
## S4 method for signature 'GRaster'
cellSize(x, mask = FALSE, lyrs = FALSE, unit = "meters2")
```

Arguments

x	A GRaster.
mask	Logical: If TRUE, then cells that are NA in x are also NA in the output.
lyrs	Logical: • If lyrs is FALSE (default), then the output has a single layer. In this case, if mask is TRUE, then cells that are NA in the *first* layer are also NA in the output. • If lyrs is TRUE, then the output has the same number of layers as x if mask is also TRUE. In this case, cells that area NA in the input will also be NA in the output in the respective layer.
unit	Character: Units of area. Partial matching is used, and case is ignored. Can be any of: • "meters2" (default), "metres2", or "m2" • "km2" or "kilometers2" • "ha" or "hectares" • "ac" or "acres" • "mi2" or "miles2" • "ft2" or "feet2"

Value

A GRaster.

See Also

terra::cellSize(), expanse(), zonalGeog(), omnibus::convertUnits()

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Elevation raster
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster:
elev <- fast(madElev)

# Cell size, no masking, single layer
cs1 <- cellSize(elev)
plot(cs1)

# Cell size, with masking, single layer
cs2 <- cellSize(elev, mask = TRUE)
plot(cs2)
```

```
# Cell size, no masking, multilayer
elev2 <- c(elev, log(elev - 200))
cs3 <- cellSize(elev2)
plot(cs3)

# Cell size, masking by 1st layer, multilayer (ignores subsequent layers)
cs4 <- cellSize(elev2, mask = TRUE)
plot(cs4)

# Cell size, masking by each layer, multilayer
cs5 <- cellSize(elev2, mask = TRUE, lyrs = TRUE)
plot(cs5)

}
```

`centroids,GVector-method`

Centroid(s) of a vector

Description

This function locates the centroid of each geometry of a `GVector`.

To use this function, you must a) have correctly specified the `addonsDir` option using `faster()`, and b) installed the **GRASS** addon `v.centerpoint`. See `addons()` and `vignette("addons", package = "fasterRaster")`.

Usage

```
## S4 method for signature 'GVector'
centroids(x, method = NULL, fail = TRUE)
```

Arguments

`x`	A `GVector`.
`method`	Character or `NULL` (default): Method used for calculating centroids. The method of calculation depends on whether the input is a `points`, `lines`, or `polygons` `GVector`. If the value is `NULL`, then the default method will be chosen, depending on the geometry type of the `GVector`: • `points`: – `"mean"` (default for `points`): Mean of coordinates. – `"median"`: Geometric median; more robust to outliers. – `"pmedian"`: Point in `x` closest to the geometric median. • `lines`: – `"mid"` (default for `lines`): Mid-point on each line; will fall exactly on the line. – `"mean"`: Center of gravity of all line segments; may not fall on the line.

  - "median: Geometric median; may not fall on the line.
- polygons:
  - "mean" (default for polygons): Center of gravity (area), calculated using area triangulation.
  - "median": Geometric mean; may not fall inside the polygon.
  - "bmedian": Geometric mean; minimum distance to boundaries; may not fall inside the polygon.

Partial matching is used and case is ignored.

fail — Logical: If TRUE (default), and the addons folder is not correctly specified, the exit the function with an error. If FALSE, then NULL will be returned with a warning.

Value

A points GVector.

See Also

terra::centroids(); **GRASS** addon module v.centerpoint.

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Points, lines, and polygons
madDypsis <- fastData("madDypsis")
madRivers <- fastData("madRivers")
madCoast4 <- fastData("madCoast4")

# Convert to  GVectors:
dypsis <- fast(madDypsis)
rivers <- fast(madRivers)
coast4 <- fast(madCoast4)

# Point centroids:
dypMean <- centroids(dypsis, fail = FALSE)
dypMedian <- centroids(dypsis, method = "median", fail = FALSE)
dypPMedian <- centroids(dypsis, method = "pmedian", fail = FALSE)

if (!is.null(dypMean)) {

plot(dypsis)
plot(dypMean, col = "red", add = TRUE)
plot(dypMedian, col = "green", pch = 2, add = TRUE)
plot(dypPMedian, col = "orange", pch = 1, add = TRUE)
legend("bottomright",
```

```
   legend = c("mean", "median", "pmedian"),
   col = c("red", "green", "orange"),
   pch = c(16, 2, 1),
   xpd = NA
)

}

# Line centroids:
riversMid <- centroids(rivers, fail = FALSE)
riversMean <- centroids(rivers, method = "mean", fail = FALSE)
riversMedian <- centroids(rivers, method = "median", fail = FALSE)

if (!is.null(riversMid)) {

plot(rivers)
plot(riversMid, col = "red", add = TRUE)
plot(riversMean, col = "green", pch = 2, add = TRUE)
plot(riversMedian, col = "orange", pch = 1, add = TRUE)
legend("bottomright",
   legend = c("mid", "mean", "median"),
   col = c("red", "green", "orange"),
   pch = c(16, 2, 1),
   xpd = NA
)

}

# Polygon centroids:
coastMean <- centroids(coast4, fail = FALSE)
coastMedian <- centroids(coast4, method = "median", fail = FALSE)
coastBMedian <- centroids(coast4, method = "bmedian", fail = FALSE)

if (!is.null(coastMean)) {

plot(coast4)
plot(coastMean, col = "red", add = TRUE)
plot(coastMedian, col = "green", pch = 2, add = TRUE)
plot(coastBMedian, col = "orange", pch = 1, add = TRUE)
legend("bottomright",
   legend = c("mean", "median", "bmedian"),
   col = c("red", "green", "orange"),
   pch = c(16, 2, 1),
   xpd = NA
)

}

}
```

`classify,GRaster-method`
Classify GRaster cell values

Description

This function classifies a 'GRaster" so that cells that have values within a given range are assigned a new value. The `subst()` function is a simpler method for replacing specific values or category levels.

Usage

```
## S4 method for signature 'GRaster'
classify(x, rcl, include.lowest = FALSE, right = TRUE, others = NULL)
```

Arguments

`x`	A `GRaster`.
`rcl`	Reclassification system: • A single integer: Number of "bins" into which to divide values. Arguments `include.lowest` and `right` apply. • A vector of numeric values: Breakpoints of bins into which to divide values. These will be sorted from lowest to highest before classification. Arguments `include.lowest` and `right` apply. • A 2-column `matrix`, `data.frame`, or `data.table`: The first column provides specific values in `x` to be replaced, and the second provides the values they are replaced with. This method is only useful for classifying `integer` `GRaster`s. Arguments `include.lowest` and `right` are ignored. Cells will be classified in the order in which values are listed in the first column. • A 3-column `matrix`, `data.frame`, or `data.table`: The first column provides the lower value of a bin, the second the upper value, and the third the value to assign to the cells in the bin. Arguments `include.lowest` and `right` apply. Cells will be classified in the order of how intervals are listed (intervals will not be sorted).
`include.lowest, right`	Logical: These arguments determine how cells that have values exactly equal to the lower or upper ends of an interval are classified. • `include.lowest` = `TRUE` and `right` = `TRUE`: All intervals will be "left-open, right-closed" except for the lowest interval, which will be "left-closed/right-closed". • `include.lowest` = `FALSE` and `right` = `FALSE`: Intervals will be "left-closed/right-open". Cells with values equal to the highest higher boundary will not be reclassified. • `include.lowest` = `TRUE` and `right` = `FALSE`: All intervals will be "left-closed/right-open", except for the highest interval, which will be "right-closed/left-closed".

- `right` = `NA`: Only useful for classifying `integer` `GRasters`. All intervals are "left-closed/right-closed". This is easier than accounting for "open" intervals when dealing with integers. Argument `include.lowest` is ignored.

`others` Integer or `NULL` (default), or `NA`: Value to assign to cells that do not fall into the set intervals. Cells with `NA` values are not reclassified. Setting `others` equal to `NULL` or `NA` replaces all other values with `NA`. The value will be coerced to an integer value.

Value

A `GRaster`. The raster will be a categorical `GRaster` if the original values were continuous (i.e., a single- or double-precision raster), or of type "integer" if the input was an integer. See `vignette("GRasters", package = "fasterRaster")`.

See Also

`terra::classify()`, `subst()`

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster:
elev <- fast(madElev)

# Classify using a scalar indicating number of bins
scalar <- classify(elev, 5)
scalar
levels(scalar)

# Classify using a vector, indicating bin break points
vector <- classify(elev, rcl = c(0, 100, 200, 300, 400, 500, 600))
vector
levels(vector)

# Classify using a 2-column matrix (only valid for integer rasters)
rcl <- data.frame(is = c(1:3, 5, 10), becomes = c(100:102, 105, 110))
twoCol <- classify(elev, rcl = rcl)
twoCol

# Classify using a 3-column table
rcl <- data.frame(
   from = c(0, 100, 200, 300, 400, 500),
   to = c(100, 200, 300, 400, 500, 600),
   becomes = c(1, 2, 3, 10, 12, 15)
```

```
)
threeCol <- classify(elev, rcl = rcl)
threeCol
levels(threeCol)

# Convert all values outside range to NA (default)
rcl <- c(100, 200, 300)
v1 <- classify(elev, rcl = rcl)
v1
plot(v1)

# Convert all values outside range to -1
rcl <- c(100, 200, 300)
v2 <- classify(elev, rcl = rcl, others = -1)
v2
plot(v2)

### Left-open/right-closed (default)
minmax(elev) # note min/max values
rcl <- c(1, 200, 570)
v3 <- classify(elev, rcl = rcl, others = 10)
levels(v3)
plot(v3)

### Left-closed/right-open
minmax(elev) # note min/max values
rcl <- c(1, 200, 570)
v4 <- classify(elev, rcl = rcl, others = 10, right = FALSE)
levels(v4)

# Left-open except for lowest bin/right-closed
minmax(elev) # note min/max values
rcl <- c(1, 200, 570)
v5 <- classify(elev, rcl = rcl, others = 10, include.lowest = TRUE)
v5 <- droplevels(v5)
levels(v5)

# Left-closed/right-open except for highest bin
minmax(elev) # note min/max values
rcl <- c(1, 200, 570)
v6 <- classify(elev, rcl = rcl, others = 10,
   right = FALSE, include.lowest = TRUE)
v6 <- droplevels(v6)
levels(v6)

}
```

---

clump,GRaster-method *Group adjacent cells with similar values*

Description

clump() identifies groups of adjacent cells that have the same value or same approximate value, and assigns them a unique number, creating "clumps" of same- or similar-valued cells.

Usage

```
## S4 method for signature 'GRaster'
clump(x, minDiff = 0, minClumpSize = 1, diagonal = TRUE)
```

Arguments

x	A GRaster.
minDiff	Numeric in the range [0, 1): Minimum difference between cells in order for them to be assigned to the same clump. This is a proportion of the range across all cells. For example, if minDiff is set to 0.01, then the maximum difference between cells in a clump can be up to 1% of the entire range across all cells. Small values can create large clumps. The default is 0, in which case values have to be exactly the same.
minClumpSize	Numeric integer >= 1. Minimum number of cells in a clump. The default is 1.
diagonal	Logical: If TRUE (default), then cells "connected" at corners will be included as part of the same clump.

Value

A GRaster.

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Elevation raster
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster:
elev <- fast(madElev)

# Find clumps based on exact values. This will appear as a gradient because
# most cells are assigned to a group of 1 cell.
exact <- clump(elev)
plot(exact)

# Clump based on approximate values:
approx <- clump(elev, minDiff = 0.0075)
plot(approx)

# Clump based on approximate values with minimum clump size:
```

```
approx20 <- clump(elev, minDiff = 0.005, minClumpSize = 20)
plot(approx20)

approx
approx20

}
```

`clusterPoints,GVector-method`
Identify clusters of points

Description

`clusterPoints()` partitions points in a "points" `GVector` into clusters.

Usage

```
## S4 method for signature 'GVector'
clusterPoints(x, method = "DBSCAN", minIn = NULL, maxDist = NULL)
```

Arguments

`x`	A "points" `GVector`.
`method`	Character: Method used to identify clusters. Explanations of methods are provided in the help page for the **GRASS** module `v.cluster`, available using `grassHelp("v.cluster")`. • `"DBSCAN"` (default): Density-Based Spatial Clustering of Applications with Noise. • `"DBSCAN2"`: A modification of DBSCAN. • `"density"`: Cluster points by relative density. • `"OPTICS"`: Ordering Points to Identify the Clustering Structure • `"OPTICS2"`: A modification of OPTICS. Case is ignored, but partial matching is not used.
`minIn`	Integer, numeric integer, or `NULL` (default): Minimum number of points in a cluster. If `NULL`, then `minIn` is set to 3 for a 2-dimensional vector and 4 for a 3-dimensional vector.
`maxDist`	Numeric or `NULL` (default): Maximum distance between neighboring points in a cluster for DBSCAN, DBSCAN2, and OPTICS. If `NULL`, the maximum distance will be set to the 99th quantile of observed pairwise distances between points.

Value

A vector of integers indicating the cluster to which each point belongs.

See Also

GRASS manual page for module `v.cluster` (see `grassHelp("v.cluster")`)

colbind,GVector-method

Add columns to the data table of a GVector

Description

colbind() adds columns to the data table of a GVector. You can combine multiple a GVector's data table with data.frames, data.tables, matrices, or the data table(s) from other GVector(s). To combine two GVectors, see rbind().

Usage

```
## S4 method for signature 'GVector'
colbind(x, ...)
```

Arguments

x, ...	The first argument must be a GVector. Subsequent arguments can be data.frames, data.tables, matrices, or GVectors. Only the data tables of subsequent GVectors are added to the table in x; the geometries are ignored.

Value

A GVector.

See Also

rbind(), addTable<-, dropTable()

Examples

```
if (grassStarted()) {

# Setup
library(sf)

# Rivers vector
madRivers <- fastData("madRivers")

# Convert sf to a GVector
rivers <- fast(madRivers)

# Convert GVector to data.frame or data.table
as.data.frame(rivers)
as.data.table(rivers)

# Subset rivers vector
rivers1 <- rivers[1:2]
rivers2 <- rivers[10:11]
```

```
# Concatenate rivers
riversCombo <- rbind(rivers1, rivers2)
riversCombo

# Add columns
newCol <- data.frame(new = 1:11)
riversCol <- colbind(rivers, newCol)
riversCol

# Remove table
riversCopy <- rivers
riversCopy # has data table
riversCopy <- dropTable(riversCopy)
riversCopy # no data table

# Add a new table
newTable <- data.frame(num = 1:11, letters = letters[1:11])
addTable(riversCopy) <- newTable
riversCopy

}
```

`combineLevels,GRaster-method`

Combine levels table from multiple categorical GRasters

Description

This function creates a single "levels" table from the levels tables of one or more categorical GRasters.

The difference between this function and `concats()` is that `concats()` creates a "combined" GRaster with a combined levels table, whereas this one just merges the levels tables.

Usage

```
## S4 method for signature 'GRaster'
combineLevels(x, ...)

## S4 method for signature 'list'
combineLevels(x, ...)
```

Arguments

`x`	A GRaster or a list of GRasters.
`...`	Arguments to pass to `data.table::merge()`.

Value

A `list` with a "levels" table (a `data.frame` or `data.table`), and the active category number for the new table. Following `terra::activeCat()`, the number is offset by 1, so a value of 1 indicates that the second column in the table should be used for the category labels, a value of 2 indicates the third column should be used, and so on.

See Also

`concats()`, terra::concats, `vignette("GRasters", package = "fasterRaster")`

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data: Land cover raster
madCover <- fastData("madCover")

# Convert categorical SpatRaster to categorical GRaster:
cover <- fast(madCover)

### Properties of categorical rasters

cover # note categories
is.factor(cover) # Is the raster categorical?
nlevels(cover) # number of levels
levels(cover) # just the value and active column
cats(cover) # all columns
minmax(cover) # min/max values
minmax(cover, levels = TRUE) # min/max categories
catNames(cover) # column names of the levels table
missingCats(cover) # categories in table with no values in raster
freq(cover) # frequency of each category (number of cells)
zonalGeog(cover) # geometric statistics

### Active column

# Which column sets the category labels?
activeCat(cover)
activeCat(cover, names = TRUE)

activeCats(c(cover, cover))

# Choose a different column for category labels:
levels(cover)
activeCat(cover) <- 2
levels(cover)

### Managing levels tables
```

```
# Remove unused levels:
nlevels(cover)
cover <- droplevels(cover)
nlevels(cover)

# Re-assign levels:
value <- c(20, 30, 40, 50, 120, 130, 140, 170)
label <- c("Cropland", "Cropland", "Forest", "Forest",
 "Grassland", "Shrubland", "Herbaceous", "Flooded")

newCats <- data.frame(value = value, label = label)

cover <- categories(cover, layer = 1, value = newCats)
cats(cover)

# This is the same as:
levels(cover) <- newCats
cats(cover)

# Are there any values not assigned a category?
missingCats(cover)

# Let's assign a category for value 210 (water):
water <- data.frame(value = 210, label = "Water")
addCats(cover) <- water
levels(cover)

# Add more information to the levels table using merge():
landType <- data.frame(
     Value = c(20, 30, 40, 50, 120),
     Type = c("Irrigated", "Rainfed", "Broadleaf evergreen",
     "Broadleaf deciduous", "Mosaic with forest")
)
cats(cover)
cover <- addCats(cover, landType, merge = TRUE)
cats(cover)

### Logical operations on categorical rasters

cover < "Forest" # 1 for cells with a value < 40, 0 otherwise
cover <= "Forest" # 1 for cells with a value < 120, 0 otherwise
cover == "Forest" # 1 for cells with value of 40-120, 0 otherwise
cover != "Forest" # 1 for cells with value that is not 40-120, 0 otherwise
cover > "Forest" # 1 for cells with a value > 120, 0 otherwise
cover >= "Forest" # 1 for cells with a value >= 120, 0 otherwise

cover %in% c("Cropland", "Forest") # 1 for cropland/forest cells, 0 otherwise

### Combine categories from different rasters

# For the example, will create a second categorical raster fromm elevation.

# Divide elevation raster into "low/medium/high" levels:
```

```
madElev <- fastData("madElev")
elev <- fast(madElev)
elev <- project(elev, cover, method = "near") # convert to same CRS
fun <- "= if(madElev < 100, 0, if(madElev < 400, 1, 2))"
elevCat <- app(elev, fun)

levs <- data.frame(
     value = c(0, 1, 2),
     elevation = c("low", "medium", "high")
)
levels(elevCat) <- list(levs)

# Combine levels:
combined <- concats(cover, elevCat)
combined
levels(combined)

# Combine levels, treating value/NA combinations as new categories:
combinedNA <- concats(cover, elevCat, na.rm = FALSE)
combinedNA
levels(combinedNA)

}
```

`Compare,GRaster,GRaster-method`

Compare-methods operations on GRasters and GRegions

Description

You can do comparative operations on `GRasters` using normal operators in **R**: <, <=, ==, !=, >=, and >. You can also use `%in%` for categorical `GRasters` (see `vignette("GRasters", package = "fasterRaster")`).

You can also compare two `GRegions` using the == and != operators. Most users of **fasterRaster** will not have to work much with "regions" (see `vignette("regions", package = "fasterRaster")`), so can ignore this functionality. `GRegions` are the same if they have the same coordinate reference system, location/project and mapset (see `vignette("projects_mapsets", package = "fasterRaster")`), topology (2D or 3D), extent, and resolution. If both are 3D, then they must also have the same vertical extent and number of depths.

Usage

```
## S4 method for signature 'GRaster,GRaster'
Compare(e1, e2)

## S4 method for signature 'logical,GRaster'
Compare(e1, e2)
```

```
## S4 method for signature 'GRaster,logical'
Compare(e1, e2)

## S4 method for signature 'numeric,GRaster'
Compare(e1, e2)

## S4 method for signature 'GRaster,numeric'
Compare(e1, e2)

## S4 method for signature 'GRaster,integer'
Compare(e1, e2)

## S4 method for signature 'integer,GRaster'
Compare(e1, e2)

## S4 method for signature 'GRaster,character'
Compare(e1, e2)

## S4 method for signature 'character,GRaster'
Compare(e1, e2)

## S4 method for signature 'GRegion,GRegion'
Compare(e1, e2)
```

Arguments

e1, e2	Values depend on the type of comparison: • Comparing GRasters to logical, numeric, character values: e1 and e2 can be any one of these. Comparison to a character string can be useful when using a categorical raster, in which case you can use something like raster1 == "Wetlands" to coerce all "wetland" cells to be 1 (TRUE) and all others 0 (FALSE) or NA (if it was originally NA). • Comparing a GRegion to another GRegion: e1 and e2 must be GRegions!

Value

Comparing GRasters: An "integer" GRaster with values of 0 (FALSE), 1 (TRUE), or NA (neither).

Comparing GRegions: Output is logical.

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")
```

```
# Convert a SpatRaster to a GRaster
elev <- fast(madElev)
elevs <- c(elev, elev, log10(elev) - 1, sqrt(elev))
names(elevs) <- c("elev1", "elev2", "log_elev", "sqrt_elev")

elev
elevs

# Comparisons
elev < 100
elev <= 100
elev == 100
elev != 100
elev > 100
elev >= 100

elev + 100 < 2 * elev

elevs > 10
10 > elevs

# logic
elev < 10 | elev > 200
elev < 10 | cos(elev) > 0.9

elev < 10 | TRUE
TRUE | elev > 200

elev < 10 | FALSE
FALSE | elev > 200

elev < 10 & cos(elev) > 0.9

elev < 10 & TRUE
TRUE & elev > 200

elev < 10 & FALSE
FALSE & elev > 200

}
```

---

compareGeom,GRaster,GRaster-method

Determine if GRasters and/or GVectors are geographically comparable

---

Description

`compareGeom()` compares geographic metadata between two or more `GRaster`s and/or `GVector`s. In many cases, spatial objects must be comparable for them to "interact" (e.g., conducting arithmetic operations, masking, etc.).

Usage

```
## S4 method for signature 'GRaster,GRaster'
compareGeom(
  x,
  y,
  ...,
  location = TRUE,
  mapset = TRUE,
  topo = TRUE,
  lyrs = FALSE,
  crs = TRUE,
  ext = TRUE,
  zext = TRUE,
  rowcol = TRUE,
  depths = TRUE,
  res = TRUE,
  zres = TRUE,
  stopOnError = TRUE,
  messages = TRUE
)

## S4 method for signature 'GVector,GVector'
compareGeom(
  x,
  y,
  ...,
  location = TRUE,
  mapset = TRUE,
  topo = FALSE,
  crs = TRUE,
  ext = FALSE,
  zext = FALSE,
  geometry = FALSE,
  stopOnError = TRUE,
  messages = TRUE
)

## S4 method for signature 'GRaster,GVector'
compareGeom(
  x,
  y,
  ...,
  location = TRUE,
  mapset = TRUE,
  topo = FALSE,
  crs = TRUE,
  ext = FALSE,
  zext = FALSE,
```

```
  stopOnError = TRUE,
  messages = TRUE
)

## S4 method for signature 'GVector,GRaster'
compareGeom(
  x,
  y,
  ...,
  location = TRUE,
  mapset = TRUE,
  topo = FALSE,
  crs = TRUE,
  ext = FALSE,
  zext = FALSE,
  stopOnError = TRUE,
  messages = TRUE
)
```

Arguments

x, y, ...	GRasters or GVectors. If y is GRaster, then the ... must also be GRasters (or missing). If y is GVector, then the ... must also be GVectors (or missing).
location, mapset	Logical: Compare **GRASS** "project/location" and "mapsets" (see vignette("projects_mapsets", package = "fasterRaster")). Default is TRUE.
topo	Logical: Test for same topology (2D or 3D). By default, this is TRUE for raster-raster comparisons, and FALSE for all others.
lyrs	Logical (rasters only): Compare number of layers of "stacked" rasters. Note this is different from number of vertical "depths" of a raster. Default is FALSE.
crs	Logical: Compare coordinate reference systems. Default is TRUE.
ext	Logical: If TRUE, test for same extent. By default, is TRUE for raster-raster comparison and FALSE for all others.
zext	Logical: Test for same vertical extents (3D only). By default, this is TRUE for raster-raster comparisons, and FALSE for all others.
rowcol	Logical (rasters only): Test for same number of rows and columns. Default is TRUE.
depths	Logical (rasters only): Test for same number of depths. Default is TRUE.
res	Logical (rasters only): Test for same resolution in x- and y-dimensions. Default is TRUE.
zres	Logical (rasters only): Test for same resolution in z dimension. Default is TRUE.
stopOnError	Logical: If TRUE (default), throw an error with an explanation if the objects are not comparable. If FALSE (default), return TRUE or FALSE.
messages	Logical: If TRUE (default), display a warning if a condition is not met. This only comes int pOnErrorisFALSE‘.
geometry	Logical (vector-vector comparison only): Compare geometry. Default is FALSE.

Value

Logical (invisibly): TRUE for no mismatches detected, FALSE for incompatibility), or side effect of throwing an error.

`complete.cases,GRaster-method`

Rows of a GRaster or GVector's table that have no NAs or that have NAs

---

Description

When applied to a categorical GRaster, compete.cases() returns TRUE for each row of the "levels" table that has no NAs in it. In contrast, missing.cases() returns TRUE for each row that has at least one NA in it. If the raster is not categorical, then NA is returned.

When applied to a GVector with a data table, complete.cases() returns TRUE for each row where there are no NAs. if the GVector has no data table, then a vector of TRUE values the same length as the total number of geometries will be returned. In contrast, missing.cases() returns TRUE for every row that has at least one NA in it. If the GVector has no data table, then a vector of FALSE values is returned.

Usage

```
## S4 method for signature 'GRaster'
complete.cases(..., levels = TRUE)

## S4 method for signature 'GVector'
complete.cases(...)

## S4 method for signature 'GRaster'
missing.cases(..., levels = TRUE)

## S4 method for signature 'GVector'
missing.cases(...)
```

Arguments

...	A GRaster or GVector.
levels	Logical (GRasters only): If TRUE (default), then assess only the "value" and activeCat() columns of the levels table (see levels()). If FALSE, then assess all columns (see cats()).

Value

Both complete.cases() and missing.cases() return the same type of object. The output depends on the input:

- A categorical GRaster with just one layer: A logical vector.

- An integer, float, or double GRaster with just one layer: NA.
- A GRaster with multiple layers: A list with one element per layer with either logical vectors or NAs, as per above.
- A GVector with a data table: A logical vector.
- A GVector without a data table: NA.

See Also

missingCats()

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Plant specimens (points) and land cover
madDypsis <- fastData("madDypsis")
madCover <- fastData("madCover")

# Convert to GVector and GRaster
dypsis <- fast(madDypsis)
cover <- fast(madCover)

### GVector

# Look at the data table:
as.data.table(dypsis)

# Which rows have no NAs?
complete.cases(dypsis)

# Which rows have at least one NA (opposite of above)?
missing.cases(dypsis)

### GRaster

# Look at the levels table:
levels(cover)

# Which rows of levels table have no NAs?
complete.cases(cover)

# Which rows have at least one NA (opposite of above)?
missing.cases(cover)

}
```

compositeRGB,GRaster-method

Combine red, green, and blue color bands to make a composite GRaster

Description

This function takes as arguments three rasters typically representing red, green, and blue color bands, and returns a single raster with values based on their combination. Typically, this raster should be plotted in grayscale.

Usage

```
## S4 method for signature 'GRaster'
compositeRGB(r, g = NULL, b = NULL, levels = 256, dither = FALSE)
```

Arguments

r, g, b	Either: • One GRaster with one band each for r, g, or b representing red, green, and blue color bands; or • r is single GRaster with 3 bands (R, G, and B bands), and g and b are NULL.
levels	Either a single value that is an integer, or a vector of integers: Number of levels of red, green, and blue intensities represented in r, g, and b. If a single value is supplied, it is assumed that all three have the same number of levels. If three values are supplied, then they correspond to the R, G, and B bands. The default is 256 (assume that R, G, and B rasters have values between 0 and 255).
dither	Logical: If TRUE, apply Floyd-Steinberg dithering. Default is FALSE.

Value

A GRaster.

See Also

plotRGB(), terra::plotRGB(), **GRASS** manual page for module r.composite (see grassHelp("r.composite"))

Examples

```
if (grassStarted()) {

# Example data
madElev <- fastData("madElev") # elevation raster
madLANDSAT <- fastData("madLANDSAT") # multi-layer raster
madRivers <- fastData("madRivers") # lines vector

# Convert SpatRaster to GRaster and SpatVector to GVector
```

```
elev <- fast(madElev)
rivers <- fast(madRivers)
landsat <- fast(madLANDSAT)

# Plot:
plot(elev)
plot(rivers, add = TRUE)

# Histograms:
hist(elev)
hist(landsat)

# Plot surface reflectance in RGB:
plotRGB(landsat, 3, 2, 1) # "natural" color
plotRGB(landsat, 4, 1, 2, stretch = "lin") # emphasize near-infrared (vegetation)

# Make composite map from RGB layers and plot in grayscale:
comp <- compositeRGB(r = landsat[[3]], g = landsat[[2]], b = landsat[[1]])
grays <- paste0("gray", 0:100)
plot(comp, col = grays)

}
```

---

concats,GRaster-method

Combine values/categories of multiple GRasters into a single GRaster

---

Description

This function takes from 2 to 10 integer or categorical (factor) GRasters and creates a single GRaster that has one value per combination of values in the inputs. For example, say that there were two input rasters, with values 1 and 2 in the one raster, and 3 and 4 in the other. If the following combinations of values occurred between the two rasters, then the output raster would be re-coded with the new values:

input_raster1	input_raster2	output_raster
1	3	0
1	4	1
2	3	2
2	4	3

If the argument na.rm is set to TRUE (which it is, by default), then whenever at least one cell has an NA value, then the output will also have an NA (i.e., a new category number is not created). However, if na.rm is FALSE, then combinations that include an NA are assigned a new category number, unless all values are NA (in which case the output will be NA).

The difference between this function and combineLevels() is that this one creates a "combined" GRaster with a combined levels table, whereas combineLevels() just merges the levels tables.

If the inputs are all categorical rasters, then a levels() table will also be returned with the new levels.

Usage

```
## S4 method for signature 'GRaster'
concats(x, ..., na.rm = TRUE)
```

Arguments

x	A GRaster with one or more layers, each of which must be have cells that represent integers or categories (factors).
...	Either missing or integer/categorical (factor) GRasters.
na.rm	Logical: If TRUE (default), then any combinations that include an NA cell will result in an NA cell in the output.

Value

A GRaster. If the inputs are all categorical (factor) rasters, then a levels table will also be returned with the new combined levels.

See Also

combineLevels(), terra::concats(), vignette("GRasters", package = "fasterRaster"), **GRASS** manual page for module r.cross (see grassHelp("r.cross"))

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data: Land cover raster
madCover <- fastData("madCover")

# Convert categorical SpatRaster to categorical GRaster:
cover <- fast(madCover)

### Properties of categorical rasters

cover # note categories
is.factor(cover) # Is the raster categorical?
nlevels(cover) # number of levels
levels(cover) # just the value and active column
cats(cover) # all columns
minmax(cover) # min/max values
minmax(cover, levels = TRUE) # min/max categories
catNames(cover) # column names of the levels table
missingCats(cover) # categories in table with no values in raster
freq(cover) # frequency of each category (number of cells)
```

```
zonalGeog(cover) # geometric statistics

### Active column

# Which column sets the category labels?
activeCat(cover)
activeCat(cover, names = TRUE)

activeCats(c(cover, cover))

# Choose a different column for category labels:
levels(cover)
activeCat(cover) <- 2
levels(cover)

### Managing levels tables

# Remove unused levels:
nlevels(cover)
cover <- droplevels(cover)
nlevels(cover)

# Re-assign levels:
value <- c(20, 30, 40, 50, 120, 130, 140, 170)
label <- c("Cropland", "Cropland", "Forest", "Forest",
 "Grassland", "Shrubland", "Herbaceous", "Flooded")

newCats <- data.frame(value = value, label = label)

cover <- categories(cover, layer = 1, value = newCats)
cats(cover)

# This is the same as:
levels(cover) <- newCats
cats(cover)

# Are there any values not assigned a category?
missingCats(cover)

# Let's assign a category for value 210 (water):
water <- data.frame(value = 210, label = "Water")
addCats(cover) <- water
levels(cover)

# Add more information to the levels table using merge():
landType <- data.frame(
     Value = c(20, 30, 40, 50, 120),
     Type = c("Irrigated", "Rainfed", "Broadleaf evergreen",
     "Broadleaf deciduous", "Mosaic with forest")
)
cats(cover)
cover <- addCats(cover, landType, merge = TRUE)
cats(cover)
```

```
### Logical operations on categorical rasters

cover < "Forest" # 1 for cells with a value < 40, 0 otherwise
cover <= "Forest" # 1 for cells with a value < 120, 0 otherwise
cover == "Forest" # 1 for cells with value of 40-120, 0 otherwise
cover != "Forest" # 1 for cells with value that is not 40-120, 0 otherwise
cover > "Forest" # 1 for cells with a value > 120, 0 otherwise
cover >= "Forest" # 1 for cells with a value >= 120, 0 otherwise

cover %in% c("Cropland", "Forest") # 1 for cropland/forest cells, 0 otherwise

### Combine categories from different rasters

# For the example, will create a second categorical raster fromm elevation.

# Divide elevation raster into "low/medium/high" levels:
madElev <- fastData("madElev")
elev <- fast(madElev)
elev <- project(elev, cover, method = "near") # convert to same CRS
fun <- "= if(madElev < 100, 0, if(madElev < 400, 1, 2))"
elevCat <- app(elev, fun)

levs <- data.frame(
     value = c(0, 1, 2),
     elevation = c("low", "medium", "high")
)
levels(elevCat) <- list(levs)

# Combine levels:
combined <- concats(cover, elevCat)
combined
levels(combined)

# Combine levels, treating value/NA combinations as new categories:
combinedNA <- concats(cover, elevCat, na.rm = FALSE)
combinedNA
levels(combinedNA)

}
```

---

connectors,GVector,GVector-method

Create lines connecting nearest features of two GVectors

Description

`connectors()` creates a lines `GVector` which represent the shortest (Great Circle) paths between each feature of one `GVector` and the nearest feature of another `GVector`.

Usage

```
## S4 method for signature 'GVector,GVector'
connectors(x, y, minDist = NULL, maxDist = NULL)
```

Arguments

x, y	GVectors.
minDist, maxDist	Either NULL (default) or numeric values: Ignore features separated by less than or greater than these distances.

Value

A GVector with a data table that has the length of each connecting line in meters.

See Also

GRASS manual for module v.distance (see grassHelp("v.distance")).

Examples

```
if (grassStarted()) {

# Setup
library(sf)

# Rivers vector and locations of Dypsis plants
madRivers <- fastData("madRivers")
madDypsis <- fastData("madDypsis")

# Convert sf's to GVectors:
dypsis <- fast(madDypsis)
rivers <- fast(madRivers)

### Connections from each point to nearest river
consFromDypsis <- connectors(dypsis, rivers)

plot(rivers, col = "blue")
plot(dypsis, add = TRUE)
plot(consFromDypsis, col = "red", add = TRUE)

### Connections from each river to nearest point
consFromRivers <- connectors(rivers, dypsis)

plot(rivers, col = "blue")
plot(dypsis, add = TRUE)
plot(consFromRivers, col = "red", add = TRUE)

}
```

convHull,GVector-method

Minimum convex hull around a spatial vector

Description

Create a minimum convex hull around a spatial vector.

Usage

```
## S4 method for signature 'GVector'
convHull(x, by = "")
```

Arguments

x	A GVector.
by	Character: If "" (default), then a convex hull is created for all geometries together. Otherwise, this is the name of a field in the vector. Hulls will be created for each set of geometries with the same value in this column.

Value

A GVector.

See Also

terra::convHull(), sf::st_convex_hull(), module v.hull in **GRASS**

Examples

```
if (grassStarted()) {

# Setup
library(sf)

# Points vector of specimens of species in the plant genus Dypsis
madDypsis <- fastData("madDypsis")

# Convert sf to a GVector:
dypsis <- fast(madDypsis)

### Convex hull for all plant specimens together:
ch <- convHull(dypsis)

### Convex hull for each species:
head(dypsis) # See the "rightsHolder" column?
chHolder <- convHull(dypsis, by = "rightsHolder")

### Plot:
```

```
plot(dypsis)
plot(ch, add = TRUE)
n <- length(chHolder)
for (i in 1:n) {
   plot(chHolder[[i]], border = i, add = TRUE)
}

}
```

`crds,GRaster-method` *Coordinates of a vector"s features or a raster"s cell centers*

Description

Returns the coordinates of the center of cells of a `GRaster` or coordinates of a `GVector`'s vertices. The output will be a `matrix`, `data.frame`, or `list`. If you want the output to be a "points" `GVector`, use `as.points()`.

Usage

```
## S4 method for signature 'GRaster'
crds(x, z = is.3d(x), na.rm = TRUE)

## S4 method for signature 'GVector'
crds(x, z = is.3d(x))

st_coordinates(x)
```

Arguments

`x`	A `GVector` or a `GRaster`.
`z`	If `TRUE` (default), return x-, y-, and z-coordinates. If `FALSE`, just return x- and y-coordinates. For 2-dimensional objects, z-coordinates will all be 0.
`na.rm`	Logical: If `TRUE`, remove cells that are `NA` (`GRaster`s only).

Value

A `matrix`, `data.frame`, or `list`.

See Also

`terra::crds()`

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Plant specimens (points), elevation (raster)
madDypsis <- fastData("madDypsis")
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster, and sf to a GVector
dypsis <- fast(madDypsis)
elev <- fast(madElev)

### Get coordinates:
dypsisPoints <- crds(dypsis)
elevPoints <- crds(elev)

head(dypsisPoints)
head(elevPoints)

}
```

---

crop,GRaster-method *Remove parts of a GRaster or GVector*

Description

`crop()` removes parts of a `GRaster` or `GVector` that fall "outside" another raster or vector. You cannot make the `GRaster` or `GVector` larger than it already is (see `extend()`). Rasters may not be cropped to the exact extent, as the extent will be enlarged to encompass an integer number of cells. If you wish to remove certain cells of a raster, see `mask()`.

Usage

```
## S4 method for signature 'GRaster'
crop(x, y, fail = TRUE)

## S4 method for signature 'GVector'
crop(x, y, extent = FALSE, fail = TRUE)
```

Arguments

`x`	A `GRaster` or `GVector` to be cropped.
`y`	A `GRaster` or `GVector` to serve as a template for cropping.
`fail`	Logical: If `TRUE` (default), and the cropped object would have zero extent in at least one dimension, then exit the function with an error. If `FALSE`, then display a warning and return `NULL`.

`extent` Logical:

- If `y` is a "points" `GVector`: Use the convex hull around `y` to crop `x`.
- If `y` is a "lines" or "polygons" `GVector`: If `TRUE`, use the extent of `y` to crop `x`.
- if `y` is a `GVector`, `x` will be cropped to the extent of `y` (`extent` is ignored).

Details

Known differences from `terra::crop()`:

- If `x` and `y` are "points" vectors, and `extent` is `TRUE`, **terra** removes points that fall on the extent boundary. **fasterRaster** does not remove points on the extent boundary.
- If `x` is a "points" vector and `y` is a "lines" vectors, and `extent` is `FALSE`, **terra** uses the extent of `y` to crop the points. **fasterRaster** uses the minimum convex hull of the lines vector.

Value

A `GRaster` or `GVector` (or `NULL` if `fail` is `FALSE` and the output would have a zero east-west and/or north-south extent).

See Also

`terra::crop()`, `sf::st_crop()`

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Elevation raster
madElev <- fastData("madElev")

# Plant specimen points vector
madDypsis <- fastData("madDypsis")

# Rivers lines vector
madRivers <- fastData("madRivers")

# Polygons vector
madCoast4 <- fastData("madCoast4")
madAnt <- madCoast4[madCoast4$NAME_4 == "Antanambe", ]

# Convert to fasterRaster format:
elev <- fast(madElev)
dypsis <- fast(madDypsis)
rivers <- fast(madRivers)
coast <- fast(madCoast4)
ant <- fast(madAnt)
```

```
### Crop raster by vector:
rastByVect <- crop(elev, ant)
plot(elev, col = "gray", legend = FALSE)
plot(rastByVect, add = TRUE)
plot(ant, add = TRUE)

### Crop raster by raster:

# For this example, make the SpatRaster smaller, then crop by this.
templateRast <- crop(madElev, madAnt)

template <- fast(templateRast)
rastByRast <- crop(elev, template)

plot(elev, col = "gray", legend = FALSE)
plot(rastByRast, add = TRUE)

### Crop vector by raster:

# For this example, make the SpatRaster smaller, then crop by this.
templateRast <- crop(madElev, madAnt)

template <- fast(templateRast)
ptsByRast <- crop(dypsis, template)

plot(elev, col = "gray", legend = FALSE)
plot(templateRast, add = TRUE)
plot(dypsis, add = TRUE)
plot(ptsByRast, pch = 21, bg = "red", add = TRUE)

### Crop vector by vector:
ptsSubset <- dypsis[1:10] # use first 10 points as cropping template

# Crop points vector by convex hull around points:
ptsByPts <- crop(dypsis, ptsSubset)
plot(dypsis)
plot(convHull(ptsSubset), lty = "dotted", border = "blue", add = TRUE)
plot(ptsByPts, col = "red", add = TRUE)
plot(ptsSubset, col = "blue", pch = 3, cex = 1.6, add = TRUE)
legend("topleft",
    legend = c("Dypsis", "Selected", "Crop template", "Convex hull"),
    pch = c(16, 16, 3, NA),
    lwd = c(NA, NA, NA, 1),
    col = c("black", "red", "blue", "blue"),
    lty = c(NA, NA, NA, "dotted"),
    xpd = NA,
    bg = "white"
)

# Crop points vector by extent of points:
ptsByPts <- crop(dypsis, ptsSubset, ext = TRUE)
plot(dypsis)
```

```
plot(ptsByPts, col = "red", add = TRUE)
plot(ptsSubset, col = "blue", pch = 3, cex = 1.6, add = TRUE)
legend("topleft",
    legend = c("Dypsis", "Selected", "Crop template"),
    pch = c(16, 16, 3),
    lwd = c(NA, NA, NA),
    col = c("black", "red", "blue"),
    lty = c(NA, NA, NA),
    xpd = NA,
    bg = "white"
)

# Crop points vector by convex hull around lines:
ptsByPts <- crop(dypsis, rivers)
plot(rivers, col = "blue", pch = 3, cex = 1.6)
plot(dypsis, add = TRUE)
plot(convHull(rivers), lty = "dotted", border = "blue", add = TRUE)
plot(ptsByPts, col = "red", add = TRUE)
legend("topleft",
    legend = c("Dypsis", "Selected", "Rivers", "Convex hull"),
    pch = c(16, 16, NA, NA),
    lwd = c(NA, NA, 1, 1),
    col = c("black", "red", "blue", "blue"),
    lty = c(NA, NA, "solid", "dotted"),
    xpd = NA,
    bg = "white"
)

# Crop points vector by extent of lines:
ptsByPts <- crop(dypsis, rivers, ext = TRUE)
plot(rivers, col = "blue", pch = 3, cex = 1.6)
plot(dypsis, add = TRUE)
plot(convHull(rivers), lty = "dotted", border = "blue", add = TRUE)
plot(ptsByPts, col = "red", add = TRUE)
legend("topleft",
    legend = c("Dypsis", "Selected", "Rivers"),
    pch = c(16, 16, NA),
    lwd = c(NA, NA, 1),
    col = c("black", "red", "blue"),
    lty = c(NA, NA, "solid"),
    xpd = NA,
    bg = "white"
)

# Crop points vector by polygon:
ptsByPts <- crop(dypsis, ant)
plot(dypsis)
plot(ant, border = "blue", pch = 3, cex = 1.6, add = TRUE)
plot(ptsByPts, col = "red", add = TRUE)
legend("topleft",
    legend = c("Dypsis", "Selected", "Antanambe"),
    pch = c(16, 16, NA),
    lwd = c(NA, NA, 1),
```

```
    col = c("black", "red", "blue"),
    lty = c(NA, NA, "solid"),
    xpd = NA,
    bg = "white"
)

# Crop lines vector by polygon:
linesByPoly <- crop(rivers, ant)
plot(rivers)
plot(ant, border = "blue", pch = 3, cex = 1.6, add = TRUE)
plot(linesByPoly, col = "red", add = TRUE)
legend("topleft",
    legend = c("Rivers", "Selected", "Antanambe"),
    col = c("black", "red", "blue"),
    lwd = 1,
    xpd = NA,
    bg = "white"
)

# Crop polygon vector by convex hull around points:
polyByPoints <- crop(ant, dypsis)
plot(dypsis, col = "red")
plot(ant, border = "blue", add = TRUE)
plot(polyByPoints, border = "red", add = TRUE)
legend("topleft",
    legend = c("Dypsis", "Antanambe", "Selected"),
    col = c("red", "blue", "red"),
    pch = c(16, NA, NA),
    lwd = c(NA, 1, 1),
    xpd = NA,
    bg = "white"
)

}
```

`crs,missing-method` *Coordinate reference system of a GRaster or GVector*

Description

Get the coordinate reference system (CRS) of a `GRaster` or `GVectors`, or from the currently active **GRASS** "project/location" (see `vignette("projects_mapsets", package = "fasterRaster")`):

- `crs()`: Return a WKT string (an object of class `character`).
- `st_crs()`: Return a `crs` object (from the **sf** package).
- `coordRef()`: Return a `list`.

Usage

```
## S4 method for signature 'missing'
crs(x)

## S4 method for signature 'GLocation'
crs(x)

## S4 method for signature 'missing'
st_crs(x, ...)

## S4 method for signature 'GLocation'
st_crs(x, ...)

st_crs(x, ...)

## S4 method for signature 'missing'
coordRef(x)

## S4 method for signature 'GRaster'
coordRef(x)

## S4 method for signature 'GVector'
coordRef(x)
```

Arguments

x	An object that inherits from a GLocation (i.e., a GRaster or GVector) or missing. If missing, the coordinate reference system of the currently active **GRASS** "project/location" is reported.
...	Other arguments (generally unused).

Value

Function crs() returns a character object, st_crs() returns crs object, and coordRef() a list.

See Also

terra::crs(), sf::st_crs()

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")
madForest2000 <- fastData("madForest2000")
```

```
# Convert SpatRasters to GRasters
elev <- fast(madElev)
forest <- fast(madForest2000)

### GRaster properties

# plotting
plot(elev)

# dimensions
dim(elev) # rows, columns, depths, layers
nrow(elev) # rows
ncol(elev) # columns
ndepth(elev) # depths
nlyr(elev) # layers

res(elev) # resolution (2D)
res3d(elev) # resolution (3D)
zres(elev) # vertical resolution
xres(elev) # vertical resolution
yres(elev) # vertical resolution
zres(elev) # vertical resolution (NA because this is a 2D GRaster)

# cell counts
ncell(elev) # cells
ncell3d(elev) # cells (3D rasters only)

# number of NA and non-NA cells
nacell(elev)
nonnacell(elev)

# topology
topology(elev) # number of dimensions
is.2d(elev) # is it 2-dimensional?
is.3d(elev) # is it 3-dimensional?

minmax(elev) # min/max values

# "names" of the object
names(elev)

# coordinate reference system
crs(elev)
st_crs(elev)
coordRef(elev)

# extent (bounding box)
ext(elev)

# vertical extent (not defined for this raster)
zext(elev)
```

```
# data type
datatype(elev) # fasterRaster type
datatype(elev, "GRASS") # GRASS type
datatype(elev, "terra") # terra type
datatype(elev, "GDAL") # GDAL type

is.integer(elev)
is.float(elev)
is.double(elev)
is.factor(elev)

# convert data type
as.int(elev) # integer; note that "elev" is already of type "integer"
as.float(elev) # floating-precision
as.doub(elev) # double-precision

# assigning
pie <- elev
pie[] <- pi # assign all cells to the value of pi
pie

# concatenating multiple GRasters
rasts <- c(elev, forest)
rasts

# subsetting
rasts[[1]]
rasts[["madForest2000"]]

# replacing
rasts[[2]] <- 2 * forest
rasts

# adding layers
rasts[[3]] <- elev > 500 # add a layer
rasts <- c(rasts, sqrt(elev)) # add another
add(rasts) <- ln(elev)
rasts

# names
names(rasts)
names(rasts) <- c("elev_meters", "2_x_forest", "high_elevation", "sqrt_elev", "ln_elev")
rasts

# remove a layer
rasts[["2_x_forest"]] <- NULL
rasts

# number of layers
nlyr(rasts)

# correlation and covariance matrices
madLANDSAT <- fastData("madLANDSAT")
```

```
landsat <- fast(madLANDSAT) # projects matrix
layerCor(landsat) # correlation
layerCor(landsat, fun = 'cov') # covariance

}
```

`datatype,GRaster-method`

Get the datatype of a GRaster or of GVector columns

Description

For `GRasters`, `datatype()` returns the data type (see `vignette("GRasters", package = "fasterRaster")`). For `GVectors`, `datatype()` returns the class of each column of the attribute table.

Usage

```
## S4 method for signature 'GRaster'
datatype(x, type = "fasterRaster", forceDouble = TRUE)

## S4 method for signature 'GVector'
datatype(x)
```

Arguments

`x`	A `GRaster` or `GVector`.
`type`	(`GRasters` only) `NULL` or character: Type of datatype to report (`GRaster` only): • `"fasterRaster"` (default): Reports the **fasterRaster** type (factor, integer, float, or double) • `"terra"`: Report the (inferred) **terra** data type (e.g., INT2U, FLT4S). Please see the table in the documentation for [writeRaster()‘ for an explanation of these codes. • `"GRASS"`: Will return "CELL" (integer), "FCELL" (floating-point value), or "DCELL" (double-floating point value) • `"GDAL"`: See GDAL: Raster Band. Please also see the table in the `writeRaster()` help page.
`forceDouble`	Logical (`GRasters` and `SpatRasters` only): If `TRUE` (default), and the raster appears to represent non-integer values, then the raster will be assumed to represent double-floating point values (**GRASS**: type "DCELL", **terra**: type "FLT8S", **fasterRaster**: type "double", and **GDAL**: type "Float64"). `forceDouble` reports the actual datatype if `type = "fasterRaster"` (i.e., the type is not forced to "double").

Value

`datatype()` for a `GRaster` returns a character. `datatype()` for a `GVector` returns a data frame, with one row per field. If the `GVector` has no attribute table, the function returns `NULL`.

See Also

terra::datatype(), vignette("GRasters", package = "fasterRaster")

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Example data
madElev <- fastData("madElev")
madForest2000 <- fastData("madForest2000")
madCoast0 <- fastData("madCoast0")
madRivers <- fastData("madRivers")
madDypsis <- fastData("madDypsis")

### GRaster properties

# convert SpatRasters to GRasters
elev <- fast(madElev)
forest <- fast(madForest2000)

# plot
plot(elev)

dim(elev) # rows, columns, depths, layers
nrow(elev) # rows
ncol(elev) # columns
ndepth(elev) # depths
nlyr(elev) # layers

res(elev) # resolution

ncell(elev) # cells
ncell3d(elev) # cells (3D rasters only)

topology(elev) # number of dimensions
is.2d(elev) # is it 2D?
is.3d(elev) # is it 3D?

minmax(elev) # min/max values

# name of object in GRASS
sources(elev)

# "names" of the object
names(elev)

# coordinate reference system
crs(elev)
```

```
# extent (bounding box)
ext(elev)

# data type
datatype(elev)

# assigning
copy <- elev
copy[] <- pi # assign all cells to the value of pi
copy

# concatenating multiple GRasters
rasts <- c(elev, forest)
rasts

# adding a raster "in place"
add(rasts) <- ln(elev)
rasts

# subsetting
rasts[[1]]
rasts[["madForest2000"]]

# assigning
rasts[[4]] <- elev > 500

# number of layers
nlyr(rasts)

# names
names(rasts)
names(rasts) <- c("elev_meters", "forest", "ln_elev", "high_elevation")
rasts

### GVector properties

# convert sf vectors to GVectors
coast <- fast(madCoast4)
rivers <- fast(madRivers)
dypsis <- fast(madDypsis)

# extent
ext(rivers)

W(rivers) # western extent
E(rivers) # eastern extent
S(rivers) # southern extent
N(rivers) # northern extent
top(rivers) # top extent (NA for 2D rasters like this one)
bottom(rivers) # bottom extent (NA for 2D rasters like this one)

# coordinate reference system
```

```
crs(rivers)
st_crs(rivers)

# column names and data types
names(coast)
datatype(coast)

# name of object in GRASS
sources(rivers)

# points, lines, or polygons?
geomtype(dypsis)
geomtype(rivers)
geomtype(coast)

is.points(dypsis)
is.points(coast)

is.lines(rivers)
is.lines(dypsis)

is.polygons(coast)
is.polygons(dypsis)

# dimensions
nrow(rivers) # how many spatial features
ncol(rivers) # hay many columns in the data frame

# number of geometries and sub-geometries
ngeom(coast)
nsubgeom(coast)

# 2- or 3D
topology(rivers) # dimensionality
is.2d(elev) # is it 2D?
is.3d(elev) # is it 3D?

# Update values from GRASS
# (Reads values from GRASS... will not appear to do anything in this case)
coast <- update(coast)

### operations on GVectors

# convert to data frame
as.data.frame(rivers)
as.data.table(rivers)

# subsetting
rivers[c(1:2, 5)] # select 3 rows/geometries
rivers[-5:-11] # remove rows/geometries 5 through 11
rivers[ , 1] # column 1
rivers[ , "NAM"] # select column
rivers[["NAM"]] # select column
```

```
rivers[1, 2:3] # row/geometry 1 and column 2 and 3
rivers[c(TRUE, FALSE)] # select every other geometry (T/F vector is recycled)
rivers[ , c(TRUE, FALSE)] # select every other column (T/F vector is recycled)

# removing data table
noTable <- dropTable(rivers)
noTable
nrow(rivers)
nrow(noTable)

# Refresh values from GRASS
# (Reads values from GRASS... will not appear to do anything in this case
# since the rivers object is up-to-date):
rivers <- update(rivers)

# Concatenating multiple vectors
rivers2 <- rbind(rivers, rivers)
dim(rivers)
dim(rivers2)

}
```

---

delaunay,GVector-method

Delaunay triangulation for points

Description

This function creates a Delaunay triangulation from a "points" GVector.

Usage

```
## S4 method for signature 'GVector'
delaunay(x)
```

Arguments

x	A GVector "points" object.

Value

A GVector.

See Also

terra::delaunay(), module v.delaunay in **GRASS**

Examples

```
if (grassStarted()) {

# Setup
library(sf)

# Example vectors
madDypsis <- fastData("madDypsis") # points
madCoast4 <- fastData("madCoast4") # polygons

# Convert sf vectors to GVectors
dypsis <- fast(madDypsis)
coast4 <- fast(madCoast4)
ant <- coast4[coast4$NAME_4 == "Antanambe"]

# Delaunay triangulation
dypsisDel <- delaunay(dypsis)
plot(dypsisDel)
plot(dypsis, pch = 1, col = "red", add = TRUE)

# Voronoi tessellation
vor <- voronoi(dypsis)
plot(vor)
plot(dypsis, pch = 1, col = "red", add = TRUE)

# Random Voronoi tessellation
rand <- rvoronoi(coast4, size = 100)
plot(rand)

}
```

---

denoise,GRaster-method

Remove or retain "noise" in a raster using PCA

Description

`denoise()` applies a principal component analysis (PCA) to layers of a `GRaster`, then uses the PCA to predict values back to a raster. This retains only coarse-scale trends, thereby removing "noise" (locally extreme values that fall far from a PC axis).

`noise()` does the opposite by first constructing the PCA, predicting values back to the raster, then subtracting these values from the original, removing coarse-scale trends and thereby leaving "noise".

Usage

```
## S4 method for signature 'GRaster'
denoise(x, scale = FALSE, percent = 80)
```

```
## S4 method for signature 'GRaster'
noise(x, scale = FALSE, percent = 80)
```

Arguments

x	A GRaster with two or more layers.
scale	Logical: If TRUE, input layers will be rescaled by dividing each layer by its overall population standard deviation. Note that rasters will always be centered (have their mean subtracted from values). Centering and scaling is recommended when rasters values are in different units. The default is FALSE (do not scale).
percent	Numeric integer or integer in the range 50 to 99 (default is 80): Minimum total variation explained in the retained PC axes. Higher values will cause denoise() to remove less noise, and noise() to return less noise. If this value to too low to retain even one axis, the function will fail with a message to that effect.

Value

A multi-layer GRaster with one layer per input.

See Also

princomp(), stats::prcomp(), **GRASS** manual page for module i.pca (see grassHelp("i.pca"))

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Climate raster:
madChelsa <- fastData("madChelsa")

# Convert a SpatRaster to a GRaster:
chelsa <- fast(madChelsa)

### Denoise:
quiet <- denoise(chelsa, scale = TRUE)

compare1 <- c(chelsa[["bio1"]], quiet[["bio1"]])
plot(compare1)

compare2 <- c(chelsa[["bio7"]], quiet[["bio7"]])
plot(compare2)

### Noise:
loud <- noise(chelsa, scale = TRUE)

compare1 <- c(chelsa[["bio1"]], loud[["bio1"]])
```

```
plot(compare1)

compare2 <- c(chelsa[["bio7"]], loud[["bio7"]])
plot(compare2)

}
```

`dim,GRegion-method` *Number of rows, columns, depths, cells, and layers*

Description

For `GRegions`: Number of rows, columns, depths, and cells:

- `dim()`: Rows and columns
- `dim3d()`: Rows, columns, and depths
- `nrow()`: Rows
- `ncol()`: Columns
- `ndepth()`: Depths (for 3-dimensional rasters only)
- `ncell()`: Number of cells (2 dimensions)
- `ncell3d()`: Number of cells (3 dimensions)

For `GRasters`: As above, plus number of layers:

- `nlyr()`: Layers (number of "stacked" rasters–different from depths of a raster).

For `GVectors`: Number of geometries and fields (columns):

- `dim()`: Number of geometries and number of columns in data table
- `nrow()`: Number of geometries
- `ncol()`: Number of columns in data table

Usage

```
## S4 method for signature 'GRegion'
dim(x)

## S4 method for signature 'missing'
dim3d(x)

## S4 method for signature 'GRegion'
dim3d(x)

## S4 method for signature 'missing'
nrow(x)

## S4 method for signature 'GRegion'
```

```
nrow(x)

## S4 method for signature 'missing'
ncol(x)

## S4 method for signature 'GRegion'
ncol(x)

## S4 method for signature 'missing'
ndepth(x)

## S4 method for signature 'GRegion'
ndepth(x)

## S4 method for signature 'missing'
ncell(x)

## S4 method for signature 'GRegion'
ncell(x)

## S4 method for signature 'missing'
ncell3d(x)

## S4 method for signature 'GRegion'
ncell3d(x)

## S4 method for signature 'GVector'
dim(x)

## S4 method for signature 'GVector'
nrow(x)

## S4 method for signature 'GVector'
ncol(x)

## S4 method for signature 'missing'
nlyr(x)

## S4 method for signature 'GRaster'
nlyr(x)
```

Arguments

x	A GRegion, GRaster, GVector, or missing. If missing, then the dimensions of the currently active "region" are returned (see vignette("regions", package = "fasterRaster")).

Value

A numeric value or vector.

See Also

ngeom(), nsubgeom(), nacell(), nonnacell(), terra::dim()

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Example data
madElev <- fastData("madElev")
madForest2000 <- fastData("madForest2000")
madCoast0 <- fastData("madCoast0")
madRivers <- fastData("madRivers")
madDypsis <- fastData("madDypsis")

### GRaster properties

# convert SpatRasters to GRasters
elev <- fast(madElev)
forest <- fast(madForest2000)

# plot
plot(elev)

dim(elev) # rows, columns, depths, layers
nrow(elev) # rows
ncol(elev) # columns
ndepth(elev) # depths
nlyr(elev) # layers

res(elev) # resolution

ncell(elev) # cells
ncell3d(elev) # cells (3D rasters only)

topology(elev) # number of dimensions
is.2d(elev) # is it 2D?
is.3d(elev) # is it 3D?

minmax(elev) # min/max values

# name of object in GRASS
sources(elev)

# "names" of the object
```

```
names(elev)

# coordinate reference system
crs(elev)

# extent (bounding box)
ext(elev)

# data type
datatype(elev)

# assigning
copy <- elev
copy[] <- pi # assign all cells to the value of pi
copy

# concatenating multiple GRasters
rasts <- c(elev, forest)
rasts

# adding a raster "in place"
add(rasts) <- ln(elev)
rasts

# subsetting
rasts[[1]]
rasts[["madForest2000"]]

# assigning
rasts[[4]] <- elev > 500

# number of layers
nlyr(rasts)

# names
names(rasts)
names(rasts) <- c("elev_meters", "forest", "ln_elev", "high_elevation")
rasts

### GVector properties

# convert sf vectors to GVectors
coast <- fast(madCoast4)
rivers <- fast(madRivers)
dypsis <- fast(madDypsis)

# extent
ext(rivers)

W(rivers) # western extent
E(rivers) # eastern extent
S(rivers) # southern extent
N(rivers) # northern extent
```

```
top(rivers) # top extent (NA for 2D rasters like this one)
bottom(rivers) # bottom extent (NA for 2D rasters like this one)

# coordinate reference system
crs(rivers)
st_crs(rivers)

# column names and data types
names(coast)
datatype(coast)

# name of object in GRASS
sources(rivers)

# points, lines, or polygons?
geomtype(dypsis)
geomtype(rivers)
geomtype(coast)

is.points(dypsis)
is.points(coast)

is.lines(rivers)
is.lines(dypsis)

is.polygons(coast)
is.polygons(dypsis)

# dimensions
nrow(rivers) # how many spatial features
ncol(rivers) # hay many columns in the data frame

# number of geometries and sub-geometries
ngeom(coast)
nsubgeom(coast)

# 2- or 3D
topology(rivers) # dimensionality
is.2d(elev) # is it 2D?
is.3d(elev) # is it 3D?

# Update values from GRASS
# (Reads values from GRASS... will not appear to do anything in this case)
coast <- update(coast)

### operations on GVectors

# convert to data frame
as.data.frame(rivers)
as.data.table(rivers)

# subsetting
rivers[c(1:2, 5)] # select 3 rows/geometries
```

```
rivers[-5:-11] # remove rows/geometries 5 through 11
rivers[ , 1] # column 1
rivers[ , "NAM"] # select column
rivers[["NAM"]] # select column
rivers[1, 2:3] # row/geometry 1 and column 2 and 3
rivers[c(TRUE, FALSE)] # select every other geometry (T/F vector is recycled)
rivers[ , c(TRUE, FALSE)] # select every other column (T/F vector is recycled)

# removing data table
noTable <- dropTable(rivers)
noTable
nrow(rivers)
nrow(noTable)

# Refresh values from GRASS
# (Reads values from GRASS... will not appear to do anything in this case
# since the rivers object is up-to-date):
rivers <- update(rivers)

# Concatenating multiple vectors
rivers2 <- rbind(rivers, rivers)
dim(rivers)
dim(rivers2)

}
```

---

disagg,GVector-method *Coerce as multipart GVector to a singlepart GVector*

Description

GVectors can contain a mix of "singlepart" and "multipart" features. A singlepart feature is a single point, set of connected line segments, or a polygon. A multipart feature is a set of lines, sets of connected line segments, or set of polygons that are treated as a single feature. This function converts all multipart features to singlepart features. If the GVector has an attribute table, rows will be duplicated so that each of the new GVector's geometries have the rows that correspond to their "parent" geometries.

If you want to "split" cells of a GRaster into smaller cells, use aggregate().

Usage

```
## S4 method for signature 'GVector'
disagg(x)
```

Arguments

x	A GVector.

Value

A GVector.

See Also

aggregate()

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")
madCoast4 <- fastData("madCoast4")

### aggregating a GRaster

# Convert:
elev <- fast(madElev)

### Aggregate GRaster by same factor in 2 dimensions
# fasterRaster
agg2 <- aggregate(elev, 2, "mean")
agg2

# Compare rasters aggregated by fasterRaster and terra.
# These should be the same.
agg2terra <- aggregate(madElev, 2)

agg2 <- rast(agg2)
agg2 <- extend(agg2, agg2terra)
agg2 - agg2terra # value is ~0

### Aggregate GRaster by a non-integer factor in 2 dimensions
# fasterRaster
agg2.9 <- aggregate(elev, 2.9, "mean")
agg2.9

# terra
agg2.9terra <- aggregate(madElev, 2.9, "mean")
agg2.9terra

# Compare rasters aggregated by fasterRaster and terra.
# These should be different.
res(agg2.9)
res(agg2.9terra) # terra rounds aggregation factor down
2 * res(madElev) # original resolution multiplied by 2

### Aggregate GRaster by different factor in 2 dimensions
```

```
agg2x3 <- aggregate(elev, c(2, 3), "mean")
agg2x3

### aggregating a GVector

madCoast4 <- fastData("madCoast4")

# Convert:
coast4 <- fast(madCoast4)

# Aggregate and disaggregate:
aggCoast <- aggregate(coast4)
disaggCoast <- disagg(coast4)

ngeom(coast4)
ngeom(aggCoast)
ngeom(disaggCoast)

# plot
oldpar <- par(mfrow = c(1, 3))
plot(coast4, main = "Original", col = 1:nrow(coast4))
plot(aggCoast, main = "Aggregated", col = 1:nrow(aggCoast))
plot(disaggCoast, main = "Disaggregated", col = 1:nrow(disaggCoast))
par(oldpar)

}
```

`distance,GRaster,missing-method`
Geographic distance

Description

This function produces a raster or a matrix of geographic distances, depending on the input:

Case 1: Argument `x` **is a** `GRaster` **and** `y` **is missing:** By default, this function replaces values in all `NA` cells with the distance between them and their closest non-`NA` cell. Alternatively, all non-`NA` cells can have their values replaced by the distance to `NA` cells. You can also specify which cells (by value) have their values replaced by distance to other cells.

Case 2: Argument `x` **is a** `GRaster` **and** `y` **is a** `GVector`**:** All cells in the raster have their value replaced by the distance to the nearest features in the `GVector`. Alternatively, calculate the distance from any cell covered by a vector object and the nearest cell *not* covered by a vector object. Note that the vector is rasterized first.

Case 3: Argument `x` **is a** `GVector` **and** `y` **is a** `GVector`**:** A matrix of pairwise distances between all features in one versus the other `GVector` is returned.

Case 4: Argument `x` **is a** `GVector` **and** `y` **is missing:** A matrix of pairwise distances between all features in the `GVector` is returned.

Usage

```
## S4 method for signature 'GRaster,missing'
distance(
  x,
  y,
  target = NA,
  fillNA = TRUE,
  unit = "meters",
  method = ifelse(is.lonlat(x), "geodesic", "Euclidean"),
  minDist = NULL,
  maxDist = NULL
)

## S4 method for signature 'GRaster,GVector'
distance(
  x,
  y,
  fillNA = TRUE,
  thick = TRUE,
  unit = "meters",
  method = ifelse(is.lonlat(x), "geodesic", "Euclidean"),
  minDist = NULL,
  maxDist = NULL
)

## S4 method for signature 'GVector,GVector'
distance(x, y, unit = "meters", minDist = NULL, maxDist = NULL)

## S4 method for signature 'GVector,missing'
distance(x, y, unit = "meters", minDist = NULL, maxDist = NULL)
```

Arguments

x	A GRaster or GVector.
y	Either missing, or a GVector.
target	Numeric: Only applicable for case 1, when x is a GRaster and y is missing. If this is NA (default), then cells with NAs have their values replaced with the distance to the nearest non-NA cell. If this is another value, then cells with these values have their values replaced with the distance to any other cell (meaning, both NA and non-NA, except for cells with this value).
fillNA	Logical: Determines which raster cells to fill with distances. • Case 1, when x is a GRaster and y is missing: If TRUE (default), fill values of NA cells with distances to non-NA cells. If FALSE, fill non-NA cells width distance to NA cells. • Case 2, when x is a GRaster and y is a GVector: If TRUE (default), then the returned raster will contain the distance from the cell to the closest feature in the vector. If FALSE, then cells covered by the vector will have their

values replaced with the distance to the nearest cell not covered, and cells that are not covered by the vector will have values of 0.

- Case 3, when x is a GVector and y is a GVector: This argument is not used in this case.

unit — Character: Units of the output. Any of:

- "meters", "metres", or "m" (default)
- "kilometers" or "km"
- "miles" or "mi"
- "nautical miles" or "nmi"
- "yards" or "yd"
- "feet" or "ft" – international, 1 foot exactly equals 0.3048 meters

Partial matching is used and case is ignored.

method — Character: The type of distance to calculate. Partial matching is used and capitalization is ignored. Possible values include:

- Euclidean (default for projected rasters): Euclidean distance.
- geodesic (default for unprojected rasters): Geographic distance. If x is unprojected (e.g., WGS84 or NAD83), then the method must be "geodesic".
- squared: Squared Euclidean distance (faster than just Euclidean distance but same rank–good for cases where only order matters).
- maximum: Maximum Euclidean distance.
- Manhattan: Manhattan distance (i.e., "taxicab" distance, distance along cells going only north-south and east-west and never along a diagonal).

minDist, maxDist — Either NULL (default) or numeric values: Ignore distances less than or greater than these distances.

thick — Logical: Only applicable for case 2, when x is a GRaster and y is a GVector. If TRUE (default), then the vector will be represented by "thickened" lines (i.e., any cell that the line/boundary touches, not just the ones on the rendering path).

Value

If x is a GRaster, then the output is a GRaster. If x is a GVector, then the output is a numeric vector.

See Also

terra::distance(); **GRASS** modules r.grow.distance and v.distance

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)
```

```
# Elevation raster, rivers vector, locations of Dypsis plants
madElev <- fastData("madElev")
madRivers <- fastData("madRivers")
madDypsis <- fastData("madDypsis")

# Convert a SpatRaster to a GRaster, and sf to a GVector
elev <- fast(madElev)
rivers <- fast(madRivers)
dypsis <- fast(madDypsis)

### case 1: GRaster by itself

# Distance between NA cells and nearest non-NA cells
naDist <- distance(elev)
names(naDist) <- "NA Distance"
plot(naDist)

# Distance between non-NA cells and nearest NA cells
nonNaDist <- distance(elev, fillNA = FALSE)
names(nonNaDist) <- "non-NA Distance"
plot(nonNaDist)

# Distance between cells with an elevation of 3 and any other cell that != 3
distFocal3 <- distance(elev, target = 3)
names(distFocal3) <- "Distance from 3"
plot(distFocal3)

# Distance between any cell and cells with a value of 3
distTo3 <- distance(elev, fillNA = FALSE, target = 3)
names(distTo3) <- "Distance to 3"
plot(distTo3)

### Case 2: GRaster and GVector
distToVect <- distance(elev, rivers)

plot(distToVect)
plot(rivers, add = TRUE)

### Case 3: GVector vs GVector
plot(rivers)
plot(dypsis, add = TRUE)

distToRivers <- distance(dypsis, rivers, unit = "yd")
distToPlants <- distance(rivers, dypsis)
distToRivers
distToPlants

### Case 4: GVector vs itself
distToItself <- distance(dypsis)
distToItself

}
```

droplevels,GRaster-method

Remove rows from the "levels" table of a categorical raster

Description

`droplevels()` removes levels (category values) from the "levels" table of a categorical `GRaster`.

Usage

```
## S4 method for signature 'GRaster'
droplevels(x, level = NULL, layer = 1)
```

Arguments

`x`	A `GRaster`.
`level`	`NULL`, character, numeric, integer, or logical: Level(s) to drop. If `NULL` (default), then all levels without values in the raster are dropped (this may remove the "levels" table entirely if all levels are dropped, converting the raster to an `integer`-type raster). If a character, this is the category label(s) to drop. If numeric or integer, this is the category value(s) to drop. If logical, values that are `TRUE` are dropped.
`layer`	Numeric integers, logical vector, or character: Layer(s) to which to add or from which to drop levels.

Value

A `GRaster`. The "levels" table of the raster is modified.

See Also

`missingCats()`, `missing.cases()`, `terra::droplevels()`, `vignette("GRasters", package = "fasterRaster")`

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data: Land cover raster
madCover <- fastData("madCover")

# Convert categorical SpatRaster to categorical GRaster:
cover <- fast(madCover)

### Properties of categorical rasters
```

```
cover # note categories
is.factor(cover) # Is the raster categorical?
nlevels(cover) # number of levels
levels(cover) # just the value and active column
cats(cover) # all columns
minmax(cover) # min/max values
minmax(cover, levels = TRUE) # min/max categories
catNames(cover) # column names of the levels table
missingCats(cover) # categories in table with no values in raster
freq(cover) # frequency of each category (number of cells)
zonalGeog(cover) # geometric statistics

### Active column

# Which column sets the category labels?
activeCat(cover)
activeCat(cover, names = TRUE)

activeCats(c(cover, cover))

# Choose a different column for category labels:
levels(cover)
activeCat(cover) <- 2
levels(cover)

### Managing levels tables

# Remove unused levels:
nlevels(cover)
cover <- droplevels(cover)
nlevels(cover)

# Re-assign levels:
value <- c(20, 30, 40, 50, 120, 130, 140, 170)
label <- c("Cropland", "Cropland", "Forest", "Forest",
 "Grassland", "Shrubland", "Herbaceous", "Flooded")

newCats <- data.frame(value = value, label = label)

cover <- categories(cover, layer = 1, value = newCats)
cats(cover)

# This is the same as:
levels(cover) <- newCats
cats(cover)

# Are there any values not assigned a category?
missingCats(cover)

# Let's assign a category for value 210 (water):
water <- data.frame(value = 210, label = "Water")
addCats(cover) <- water
```

```
levels(cover)

# Add more information to the levels table using merge():
landType <- data.frame(
     Value = c(20, 30, 40, 50, 120),
     Type = c("Irrigated", "Rainfed", "Broadleaf evergreen",
     "Broadleaf deciduous", "Mosaic with forest")
)
cats(cover)
cover <- addCats(cover, landType, merge = TRUE)
cats(cover)

### Logical operations on categorical rasters

cover < "Forest" # 1 for cells with a value < 40, 0 otherwise
cover <= "Forest" # 1 for cells with a value < 120, 0 otherwise
cover == "Forest" # 1 for cells with value of 40-120, 0 otherwise
cover != "Forest" # 1 for cells with value that is not 40-120, 0 otherwise
cover > "Forest" # 1 for cells with a value > 120, 0 otherwise
cover >= "Forest" # 1 for cells with a value >= 120, 0 otherwise

cover %in% c("Cropland", "Forest") # 1 for cropland/forest cells, 0 otherwise

### Combine categories from different rasters

# For the example, will create a second categorical raster fromm elevation.

# Divide elevation raster into "low/medium/high" levels:
madElev <- fastData("madElev")
elev <- fast(madElev)
elev <- project(elev, cover, method = "near") # convert to same CRS
fun <- "= if(madElev < 100, 0, if(madElev < 400, 1, 2))"
elevCat <- app(elev, fun)

levs <- data.frame(
     value = c(0, 1, 2),
     elevation = c("low", "medium", "high")
)
levels(elevCat) <- list(levs)

# Combine levels:
combined <- concats(cover, elevCat)
combined
levels(combined)

# Combine levels, treating value/NA combinations as new categories:
combinedNA <- concats(cover, elevCat, na.rm = FALSE)
combinedNA
levels(combinedNA)

}
```

dropRows,data.table-method

Remove rows in a data.table, data.frame, or matrix.

Description

As of September of 2023, the **data.table** package does not have a function for removing rows by index. This function does this job.

Usage

```
## S4 method for signature 'data.table'
dropRows(x, drops)

## S4 method for signature 'data.frame'
dropRows(x, drops)

## S4 method for signature 'matrix'
dropRows(x, drops)
```

Arguments

x	A data.table or data.frame.
drops	Numeric, integer, or logical vector: Indices or indicators of rows to remove. If a logical vector is supplied, rows that correspond to TRUE will be removed. If the vector is shorter than the number of rows, values of drops will be recycled.

Value

A data.table or data.frame.

Examples

```
library(data.table)

dt <- data.table(
   x = 1:10,
   y = letters[1:10],
   z = rnorm(10)
)

# make some values NA
dt[x == 4 | x == 8, y := NA_character_]
dt

# Replace NAs:
replaceNAs(dt, replace = -99, cols = "y")
dt
```

```
# Drop rows:
dropped <- dropRows(dt, 8:10)
dropped

# NB May not print... in that case, use:
print(dropped)

# We can also use replaceNAs() on vectors:
y <- 1:10
y[c(2, 10)] <- NA
replaceNAs(y, -99)

# Same as:
y <- 1:10
y[c(2, 10)] <- NA
y[is.na(y)] <- -99
```

`erase,GVector,GVector-method` *Select parts of a polygon GVector erase shared by another polygon GVector*

Description

The `erase()` function removes from the `x` "polygons" `GVector` parts that overlap with the `y` "polygons" `GVector`. You can also use the `-` operator (e.g., `vect1 - vect2`).

Usage

```
## S4 method for signature 'GVector,GVector'
erase(x, y)
```

Arguments

`x`, `y`	`GVectors`.

Value

A `GVector`.

See Also

`c()`, `aggregate()`, `crop()`, `intersect()`, `union()`, `xor()`

Examples

```
if (grassStarted()) {

# Setup
library(sf)

# Polygon of coastal Madagascar and Dypsis specimens
madCoast4 <- fastData("madCoast4") # polygons
madDypsis <- fastData("madDypsis") # points

# Convert vectors:
coast4 <- fast(madCoast4)
dypsis <- fast(madDypsis)

# Create another polygons vector from a convex hull around Dypsis points
hull <- convHull(dypsis)

### union()

unioned <- union(coast4, hull)
plot(unioned)

plus <- coast4 + hull # same as union()

### intersect

inter <- intersect(coast4, hull)
plot(coast4)
plot(hull, border = "red", add = TRUE)
plot(inter, border = "blue", add = TRUE)

### xor

xr <- xor(coast4, hull)
plot(coast4)
plot(xr, border = "blue", add = TRUE)

### erase

erased <- erase(coast4, hull)
plot(coast4)
plot(erased, border = "blue", add = TRUE)

minus <- coast4 - hull # same as erase()

}
```

`expanse,GVector-method`

Area of polygons or length of lines

Description

This function calculates the area of each polygon in a "polygons" GVector or the length of lines in a "lines" GVector.

Usage

```
## S4 method for signature 'GVector'
expanse(x, unit = "m")
```

Arguments

x	A "polygons" or "lines" GVector.
unit	Character: Units in which to report values. Areal units are squared, linear are not. Can be any of: • "meters"(default), "metres", or "m" • "km" or "kilometers" • "ha" or "hectares" • "ft" or "feet" • "ac" or "acres" • "percent" Partial matching is used and case is ignored.

Value

Numeric values, one per geometry in x.

Examples

```
if (grassStarted()) {

# Setup
library(sf)

# Example data:
madCoast4 <- fastData("madCoast4")
madRivers <- fastData("madRivers")
madDypsis <- fastData("madDypsis")

# Convert sf vectors to GVectors:
coast <- fast(madCoast4)
rivers <- fast(madRivers)
dypsis <- fast(madDypsis)

# Geographic properties:
ext(rivers) # extent
crs(rivers) # coordinate reference system
st_crs(rivers) # coordinate reference system
coordRef(rivers) # coordinate reference system
```

```
# Column names and data types:
names(coast)
datatype(coast)

# Points, lines, or polygons?
geomtype(dypsis)
geomtype(rivers)
geomtype(coast)

is.points(dypsis)
is.points(coast)

is.lines(rivers)
is.lines(dypsis)

is.polygons(coast)
is.polygons(dypsis)

# Number of dimensions:
topology(rivers)
is.2d(rivers) # 2-dimensional?
is.3d(rivers) # 3-dimensional?

# Just the data table:
as.data.frame(rivers)
as.data.table(rivers)

# Top/bottom of the data table:
head(rivers)
tail(rivers)

# Vector or table with just selected columns:
names(rivers)
rivers$NAME
rivers[[c("NAM", "NAME_0")]]
rivers[[c(3, 5)]]

# Select geometries/rows of the vector:
nrow(rivers)
selected <- rivers[2:6]
nrow(selected)

# Plot:
plot(coast)
plot(rivers, col = "blue", add = TRUE)
plot(selected, col = "red", lwd = 2, add = TRUE)

# Vector math:
hull <- convHull(dypsis)

un <- union(coast, hull)
sameAsUnion <- coast + hull
plot(un)
```

```
plot(sameAsUnion)

inter <- intersect(coast, hull)
sameAsIntersect <- coast * hull
plot(inter)
plot(sameAsIntersect)

er <- erase(coast, hull)
sameAsErase <- coast - hull
plot(er)
plot(sameAsErase)

xr <- xor(coast, hull)
sameAsXor <- coast / hull
plot(xr)
plot(sameAsXor)

# Vector area and length:
expanse(coast, unit = "km") # polygons areas
expanse(rivers, unit = "km") # river lengths

### Fill holes

# First, we will make some holes by creating buffers around points.
buffs <- buffer(dypsis, 500)

holes <- coast - buffs
plot(holes)

filled <- fillHoles(holes, fail = FALSE)

}
```

`ext,missing-method` *Spatial bounds of a GRaster or GVector*

Description

These functions return the extent of a `GSpatial` object (`GRegions`, `GRasters`, and `GVectors`):

- `ext()`: 2-dimensional spatial extent (i.e., westernmost/easternmost and southernmost/northernmost coordinates of area represented).
- `zext()`: Vertical extent (i.e., topmost and bottom-most elevation of the volume represented). The vertical extent is not `NA` only if the object is 3-dimensional.
- `W()`, `E()`, `N()`, `S()`: Coordinates of one side of horizontal extent.
- `top()` and `bottom()`: Coordinates of top and bottom of vertical extent.

Usage

```
## S4 method for signature 'missing'
ext(x, vector = FALSE)

## S4 method for signature 'GSpatial'
ext(x, vector = FALSE)

## S4 method for signature 'missing'
zext(x)

## S4 method for signature 'GSpatial'
zext(x)

## S4 method for signature 'missing'
W(x, char = FALSE)

## S4 method for signature 'GSpatial'
W(x, char = FALSE)

## S4 method for signature 'missing'
E(x, char = FALSE)

## S4 method for signature 'GSpatial'
E(x, char = FALSE)

## S4 method for signature 'missing'
N(x, char = FALSE)

## S4 method for signature 'GSpatial'
N(x, char = FALSE)

## S4 method for signature 'missing'
S(x, char = FALSE)

## S4 method for signature 'GSpatial'
S(x, char = FALSE)

## S4 method for signature 'missing'
top(x, char = FALSE)

## S4 method for signature 'GSpatial'
top(x, char = FALSE)

## S4 method for signature 'GSpatial'
bottom(x, char = FALSE)

## S4 method for signature 'GSpatial'
bottom(x, char = FALSE)
```

Arguments

x	An object that inherits from GSpatial (i.e., a GRaster or GVector) or missing. If missing, then the horizontal or vertical extent of the currently active "region" is returned (see vignette("regions", package = "fasterRaster")).
vector	Logical: If FALSE (default), return a SpatExtent object. If TRUE, return the extent as a named vector.
char	Logical: If FALSE (default), return a numeric value. If TRUE, return as a character.

Value

The returned values depend on the function:

- ext(): A SpatExtent object (**terra** package) or a numeric vector.
- zext(): A numeric vector.
- W(), E(), N(), S(), top(), and bottom(): A numeric value or character.

See Also

terra::ext(), sf::st_bbox()

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Example data
madElev <- fastData("madElev")
madForest2000 <- fastData("madForest2000")
madCoast0 <- fastData("madCoast0")
madRivers <- fastData("madRivers")
madDypsis <- fastData("madDypsis")

### GRaster properties

# convert SpatRasters to GRasters
elev <- fast(madElev)
forest <- fast(madForest2000)

# plot
plot(elev)

dim(elev) # rows, columns, depths, layers
nrow(elev) # rows
ncol(elev) # columns
ndepth(elev) # depths
nlyr(elev) # layers
```

```
res(elev) # resolution

ncell(elev) # cells
ncell3d(elev) # cells (3D rasters only)

topology(elev) # number of dimensions
is.2d(elev) # is it 2D?
is.3d(elev) # is it 3D?

minmax(elev) # min/max values

# name of object in GRASS
sources(elev)

# "names" of the object
names(elev)

# coordinate reference system
crs(elev)

# extent (bounding box)
ext(elev)

# data type
datatype(elev)

# assigning
copy <- elev
copy[] <- pi # assign all cells to the value of pi
copy

# concatenating multiple GRasters
rasts <- c(elev, forest)
rasts

# adding a raster "in place"
add(rasts) <- ln(elev)
rasts

# subsetting
rasts[[1]]
rasts[["madForest2000"]]

# assigning
rasts[[4]] <- elev > 500

# number of layers
nlyr(rasts)

# names
names(rasts)
names(rasts) <- c("elev_meters", "forest", "ln_elev", "high_elevation")
rasts
```

```
### GVector properties

# convert sf vectors to GVectors
coast <- fast(madCoast4)
rivers <- fast(madRivers)
dypsis <- fast(madDypsis)

# extent
ext(rivers)

W(rivers) # western extent
E(rivers) # eastern extent
S(rivers) # southern extent
N(rivers) # northern extent
top(rivers) # top extent (NA for 2D rasters like this one)
bottom(rivers) # bottom extent (NA for 2D rasters like this one)

# coordinate reference system
crs(rivers)
st_crs(rivers)

# column names and data types
names(coast)
datatype(coast)

# name of object in GRASS
sources(rivers)

# points, lines, or polygons?
geomtype(dypsis)
geomtype(rivers)
geomtype(coast)

is.points(dypsis)
is.points(coast)

is.lines(rivers)
is.lines(dypsis)

is.polygons(coast)
is.polygons(dypsis)

# dimensions
nrow(rivers) # how many spatial features
ncol(rivers) # hay many columns in the data frame

# number of geometries and sub-geometries
ngeom(coast)
nsubgeom(coast)

# 2- or 3D
topology(rivers) # dimensionality
```

```
is.2d(elev) # is it 2D?
is.3d(elev) # is it 3D?

# Update values from GRASS
# (Reads values from GRASS... will not appear to do anything in this case)
coast <- update(coast)

### operations on GVectors

# convert to data frame
as.data.frame(rivers)
as.data.table(rivers)

# subsetting
rivers[c(1:2, 5)] # select 3 rows/geometries
rivers[-5:-11] # remove rows/geometries 5 through 11
rivers[ , 1] # column 1
rivers[ , "NAM"] # select column
rivers[["NAM"]] # select column
rivers[1, 2:3] # row/geometry 1 and column 2 and 3
rivers[c(TRUE, FALSE)] # select every other geometry (T/F vector is recycled)
rivers[ , c(TRUE, FALSE)] # select every other column (T/F vector is recycled)

# removing data table
noTable <- dropTable(rivers)
noTable
nrow(rivers)
nrow(noTable)

# Refresh values from GRASS
# (Reads values from GRASS... will not appear to do anything in this case
# since the rivers object is up-to-date):
rivers <- update(rivers)

# Concatenating multiple vectors
rivers2 <- rbind(rivers, rivers)
dim(rivers)
dim(rivers2)

}
```

`extend,GRaster,numeric-method`

Add rows and columns around a writeRaster

Description

`extend()` adds cells around a raster, making it larger.

Usage

```
## S4 method for signature 'GRaster,numeric'
extend(x, y, fill = NA)

## S4 method for signature 'GRaster,SpatRaster'
extend(x, y, snap = "near", fill = NA)

## S4 method for signature 'GRaster,SpatVector'
extend(x, y, snap = "near", fill = NA)

## S4 method for signature 'GRaster,SpatExtent'
extend(x, y, snap = "near", fill = NA)

## S4 method for signature 'GRaster,sf'
extend(x, y, snap = "near", fill = NA)

## S4 method for signature 'GRaster,GSpatial'
extend(x, y, snap = "near", fill = NA)
```

Arguments

x	A GRaster.
y	Any of: • An object from which an extent can be obtained; i.e., a SpatRaster, SpatVector, SpatExtent, sf vector, or a GSpatial object (any of GRaster, GVector, or GRegion). If the extent of x is "outside" the extent of y on any side, the side(s) of x that are outside will be kept as-is (i.e., the extent of x will never be shrunk). • A single positive integer: Number of rows and columns to add to the top, bottom, and sides of the raster. • Two integers >= 0: Number of columns (1st value) to add to the sides, and number of rows (2nd value) to add to the top and bottom of the raster. • Four integers >= 0: Number of rows and columns to add (left column, right column, bottom row, top row).
fill	Numeric: Value to place in the new cells. The default is NA.
snap	Character: Method used to align y to x. Partial matching is used. This is only used if y is not a set of numbers. • "near" (default): Round to nearest row/column • "in": Round "inward" toward the extent of x to nearest row/column • "out": Round "outward" away from the extent of x to the nearest row/column.

Details

Known issues: When GRasters are saved to disk explicitly using writeRaster(), or implicitly using rast() or plot(), rows and columns that are entirely NA are dropped.

Value

A GRaster.

See Also

terra::extend()

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")
madRivers <- fastData("madRivers")

# Send spatial objects to GRASS:
elev <- fast(madElev)
rivers <- fast(madRivers)

# Extend raster by number of rows/columns:
extended1 <- extend(elev, 10, fill = 900)
extended2 <- extend(elev, c(10, 20), fill = 900)
extended3 <- extend(elev, c(10, 80, 0, 100), fill = 900)
dim(elev)
dim(extended1)
dim(extended2)
dim(extended3)

plot(extended3)

# When exporting a raster, NA rows and columns are removed.
extended4 <- extend(elev, 100, fill=1) # default fill is NA
extended4terra <- rast(extended4)

dim(extended4)
dim(extended4terra)

plot(extended4)

# Extend the raster by another object with a wider extent.

# For tis example, first crop the raster, then extend it.
elevCrop <- crop(elev, rivers)
uncrop <- extend(elevCrop, elev, fill = 900)
plot(uncrop)

}
```

`extract,GRaster,GVector-method`
Extract values from a GRaster at locations in a points GVector

Description

`extract()` obtains the values of a `GRaster` or `GVector` associated with the locations of a set of points. The output depends on the input:

- **Case #1:** `x` **is a numeric or integer** `GRaster` **and** `y` **is a points** `GVector`: Returns values of cells that have points. If `xy` is `TRUE`, also returns the coordinates of the points.
- **Case #2:** `x` **is a categorical (factor)** `GRaster` **and** `y` **is a points** `GVector`: Same as case #1, but if `cats` is `TRUE`, returns category labels of cells that have points. If `xy` is `TRUE`, also returns the coordinates of the points.
- **Case #3:** `x` **is a categorical** `GRaster` **and** `y` **is a lines or polygons** `GVector`: Returns a summary (e.g., mean, standard deviation, etc.) of all cells that overlap the line(s) or polygon(s).
- **Case #4:** `x` **is a** `GVector` **and** `y` **is a points** `GVector`: Returns the data table row associated each point. If `xy` is `TRUE`, also returns the coordinates of the points. Note that whenever a points `GVector` is allowed for `y`, a `data.frame`, `data.table`, `matrix`, or `numeric` values representing points can be used instead.

Usage

```
## S4 method for signature 'GRaster,GVector'
extract(
  x,
  y,
  fun = "mean",
  prob = 0.5,
  overlap = TRUE,
  xy = FALSE,
  cats = TRUE,
  verbose = FALSE,
  ...
)

## S4 method for signature 'GRaster,data.frame'
extract(x, y, xy = FALSE, cats = TRUE)

## S4 method for signature 'GRaster,data.table'
extract(x, y, xy = FALSE, cats = TRUE)

## S4 method for signature 'GRaster,matrix'
extract(x, y, xy = FALSE, cats = TRUE)

## S4 method for signature 'GRaster,numeric'
```

```
extract(x, y, xy = FALSE, cats = TRUE)

## S4 method for signature 'GVector,GVector'
extract(x, y, xy = FALSE, verbose = TRUE)

## S4 method for signature 'GVector,data.frame'
extract(x, y, xy = FALSE, verbose = TRUE)

## S4 method for signature 'GVector,data.table'
extract(x, y, xy = FALSE, verbose = TRUE)

## S4 method for signature 'GVector,matrix'
extract(x, y, xy = FALSE, verbose = TRUE)

## S4 method for signature 'GVector,numeric'
extract(x, y, xy = FALSE)
```

Arguments

x	A GRaster or GVector.
y	A GVector, *or* a data.frame or matrix where the first two columns represent longitude and latitude (in that order), *or* a two-element numeric vector where the first column represents longitude and the second latitude. Values of x will be extracted from the points in y. GVectors can be of types points, lines, or polygons.
fun	Character vector: Name(s) of function(s) to apply to values. This is used when x is a GRaster and y is a lines or polygons GVector. The method(s) specified by fun will be applied to all cell values that overlap with each geometry (i.e., individual cell values will not be returned). Valid functions include: • "countNonNA": Number of overlapping cells. • "countNA": Number of overlapping NA cells. • "mean": Average. • "min": Minimum. • "max": Minimum. • "sum": Sum. • "range": Maximum - minimum. • "sd": Sample standard deviation (same as stats::sd()). • "sdpop": Population standard deviation. • "var": Sample variance (same as stats::var()). • "varpop": Population variance. • "cv": Coefficient of variation. • "cvpop": Population coefficient of variation. • "median": Median. • "quantile": Quantile; you can specify the quantile using the prob argument.

prob	Numeric in the range from 0 to 1: Quantile which to calculate. The value of prob will be rounded to the nearest hundredth.
overlap	Logical: If TRUE (default), and y is a lines or polygons GVector, then account for potential overlap of geometries when extracting. This can be slow, so if you are sure geometries do not overlap, you can change this to FALSE. This argument is ignored if y is a points GVector.
xy	Logical: If TRUE and y represents points, also return the coordinates of each point. Default is FALSE.
cats	Logical (extracting from a raster): If TRUE (default) and x is a categorical GRaster, then return the category labels instead of the values.
verbose	Logical: If TRUE, display progress (will only function when extracting from points on a GRaster when the number of GRasters is large, or when extracting using a "points" GVector with lots of points).
...	Arguments to pass to project(). This is used only if extracting from a GRaster at locations specified by a GVector, and they have a different coordinate reference system. In this case, users should specify the wrap argument to project().

Value

A data.frame or data.table.

See Also

terra::extract(), and modules r.what and v.what in **GRASS**

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Example data: elevation raster and points vector
madElev <- fastData("madElev") # raster
madCover <- fastData("madCover") # categorical raster
madDypsis <- fastData("madDypsis") # points vector
madRivers <- fastData("madRivers") # lines vector
madCoast4 <- fastData("madCoast4") # polygons vector

# Convert to fasterRaster formats:
elev <- fast(madElev) # raster
cover <- fast(madCover) # categorical raster
dypsis <- fast(madDypsis) # points vector
rivers <- fast(madRivers) # lines vector
coast <- fast(madCoast4) # polygons vector

# Get values of elevation at points where Dypsis species are located:
extract(elev, dypsis, xy = TRUE)
```

```
# Extract from categorical raster at points:
categories <- extract(cover, dypsis)
categoryValues <- extract(cover, dypsis, cats = FALSE)
categories
categoryValues

# Extract and summarize values on a raster across polygons:
extract(elev, coast, fun = c("sum", "mean", "countNonNA"), overlap = FALSE)

# Extract and summarize values on a raster across lines:
extract(elev, rivers, fun = c("sum", "mean", "countNonNA"), overlap = FALSE)

# Extract from a polygons vector at a points vector:
polysFromPoints <- extract(coast, dypsis, xy = TRUE)
head(polysFromPoints) # first 3 are outside polygons vector, next 3 are inside

}
```

`fast` *Create a GRaster or GVector*

Description

`fast()` creates a `GRaster` or `GVector` from 1) a file; 2) from a `SpatRaster`, `SpatVector`, or `sf` vector; or 3) from a numeric vector, `matrix`, `data.frame`, or `data.table`. Behind the scenes, this function will also create a connection to **GRASS** if none has yet been made yet.

GRASS supports loading from disk a variety of raster formats (see the **GRASS** manual page for `r.in.gdal` (see `grassHelp("r.in.gdal")`) and vector formats `v.in.ogr` (see grassHelp("v.in.ogr")‘), though not all of them will work with this function.

Note that `GVectors` may fail to be created if they contain issues that do not coincide with the topological data model used by **GRASS**. The most common of these is overlapping polygons. See *Details* on how to fix these kinds of issues.

Note also that **GRASS** (and thus, **fasterRaster**) is *not* very fast when loading vectors. So, if the vector is large and you only want a portion of it, consider using the `extent` argument to load the spatial subset you need.

Usage

```
## S4 method for signature 'character'
fast(
  x,
  rastOrVect = NULL,
  levels = TRUE,
  extent = NULL,
  correct = TRUE,
  snap = NULL,
  area = NULL,
```

```
  steps = 10,
  dropTable = FALSE,
  resolve = NA,
  verbose = TRUE,
  ...
)

## S4 method for signature 'SpatRaster'
fast(x, ...)

## S4 method for signature 'SpatVector'
fast(
  x,
  extent = NULL,
  correct = TRUE,
  snap = NULL,
  area = NULL,
  steps = 10,
  dropTable = FALSE,
  resolve = NA,
  verbose = TRUE
)

## S4 method for signature 'sf'
fast(
  x,
  extent = NULL,
  correct = TRUE,
  snap = NULL,
  area = NULL,
  steps = 10,
  resolve = NA,
  dropTable = FALSE,
  verbose = TRUE
)

## S4 method for signature 'missing'
fast(x, rastOrVect, crs = "")

## S4 method for signature 'numeric'
fast(x, crs = "", keepgeom = FALSE)

## S4 method for signature 'data.frame'
fast(x, geom = 1:2, crs = "", keepgeom = FALSE)

## S4 method for signature 'data.table'
fast(x, geom = 1:2, crs = "", keepgeom = FALSE)
```

```
## S4 method for signature 'matrix'
fast(x, geom = 1:2, crs = "", keepgeom = FALSE)
```

Arguments

x	Any one of: • A SpatRaster raster. Rasters can have one or more layers. • A SpatVector or sf spatial vector. See especially arguments correct, area, snap, steps, and verbose. • A character string or a vector of strings with the path(s) and filename(s) of one or more rasters or one vector to be loaded directly into **GRASS**. The function will attempt to ascertain the type of object from the file extension (raster or vector), but it can help to indicate which it is using the rastOrVect argument if it is unclear. For rasters, see especially argument levels. For vectors, see especially arguments correct, resolve, area, snap, steps, and verbose. • A vector with an even number of numeric values representing longitude/latitude pairs. See arguments geom, keepgeom, and crs. • A data.frame, data.table, or matrix: Create a points GVector. Two of the columns must represent longitude and latitude. See arguments geom, keepgeom, and crs. • Missing: Creates a generic GRaster or GVector. You must specify rastOrVect; for example, fast(rastOrVect = "raster"). Also see argument crs.
rastOrVect	Either NULL (default), or "raster" or "vector": If x is a filename, then the function will try to ascertain whether it represents a raster or a vector, but sometimes this will fail. In that case, it can help to specify if the file holds a raster or vector. Partial matching is used.
levels	(GRasters only): Any of: • Logical: If TRUE (default) and at least one layer of a raster is of type integer, search for a "levels" file, load it, and attach levels. A levels file will have the same name as the raster file, but end with any of "rdata", "rdat", "rda", "rds", "csv", or "tab" (case will generally not matter). If such a file is not found, no levels will be assigned. The levels file must contain either a data.frame, data.table, or list of data.frames or data.tables, or NULL. • A data.frame, data.table, or list of data.frames or data.tables with categories for categorical rasters. The first column of a table corresponds to raster values and must be of type integer. A subsequent column corresponds to category labels. By default, the second column is assumed to represent labels, but this can be changed with activeCat<-. Level tables can also be NULL (e.g., data.fame(NULL)). You can also assign levels after loading a raster using levels<-. • NULL: Do not attach a levels table. #'
extent	(GVectors only): Either a NULL (default), or a GVector, a SpatVector, a SpatExtent object, an sf vector, a bbox object, or a numeric vector of 4 values providing a bounding box. If provided, only vector features within this bounding box are

imported. If `extent` is a numeric vector, the values *must* be in the order west, east, south, north. If `NULL`, the entire vector is imported.

`correct` Logical (`GVectors` only): Correct topological issues. See *Details* for more details! By default, this is `TRUE`.

`snap` `GVectors` only: Numeric or `NULL` (default). The value of `snap` indicates how close vertices need to be for them to be shifted to to the same location. Units of `snap` are map units (usually meters), or degrees for unprojected CRSs. For lines and polygons vectors, a value of `NULL` will invoke an iterative procedure to find an optimal, smallest value of `snap`. To turn snapping off, set `snap = 0`. See *Details* for more details!

`area` Polygon `GVectors` only: Either a positive numeric value or `NULL` (default). Remove polygons with an area smaller than this value. Units of `area` are in square meters (regardless of the CRS). If `NULL`, then an iterative procedure is used to identify a value of `area` that results in a topologically correct polygon vector. For point and lines vectors, this argument is ignored. To turn area removal off, set `area = 0`. See *Details* for more details!

`steps` `GVectors` only: A positive integer > 1 (default is 10). When using automatic vector correction (i.e., either `snap = NULL` and/or `area = NULL`), this is the number of values of `snap` and/or `area` to try to generate a correct topology, including no snapping or polygon removal (i.e, `snap = 0` and `area = 0`).

`dropTable` `GVectors` only: Logical. If `TRUE`, then drop the data table associated with a vector. By default, this is `FALSE`. See *Details* for more details!

`resolve` `GVectors` only: Character. If a `GVector` would be topologically invalid after a first attempt at creating it, then this method will be used to resolve the issue and create a valid `GVector`. Partial matching is used.

- `"disaggregate"`: Coerce each area of overlap between polygons into its own geometry. The output will not have a data table associated with it.
- `"aggregate"`: Coerce all geometries into a "multipart" geometry so it acts like a single geometry. The output will not have a data table associated with it.
- `NA` (default): Do neither of the above and if either `snap` or `area` is `NULL`, keep trying to create the `GVector`. Upon success, the `GVector` will retain any data table associated with it unless `dropTable` is `FALSE`.

`verbose` `GVectors` only: Logical. Displays progress when using automatic topology correction.

`...` Other arguments::

- `table` (`GVectors`–useful mainly to developers, not most users): A `data.frame` or `data.table` with one row per geometry in a `GVector`. Serves as an attribute table.
- `xVect` (`GVectors`–useful mainly to developers, not most users): The `SpatVector` that corresponds to the file named by `x`.

`crs` String: Coordinate reference system (CRS) WKT2 string. This argument is used for creating a `GVector` from a `numeric` vector or a `data.frame` or similar, or from `fast(rastOrVect = "vector")` or `fast(rastOrVect = "raster")`. By default, the function will use the value of `crs()` (no arguments), which is the

CRS of the current **GRASS** "project/location" (see `vignette("projects_mapsets", package = "fasterRaster")`).

`keepgeom` Logical: If `x` is a set of `numeric` coordinates, or a `data.frame` or similar, then they can be coerced into a `points GVector`. If `keepgeom` is `TRUE`, then the coordinates will be included in the data table of the `GVector`. The default is `FALSE`.

`geom` Character or integer vector: If `x` is a `data.frame`, `data.table`, or `matrix`, this specifies which columns of `x` represent longitude and latitude. Columns can be given by name (a character vector) or index (a numeric or integer vector). The default is to use the first two columns of `x`.

Details

GRASS uses a "topological" model for vectors. Topological issues generally arise only with polygon vectors, not point or line vectors. Sometimes, polygons created in other software are topologically incorrect–the borders of adjacent polygons may cross one another, or there may be small gaps between them. These errors can be corrected by slightly shifting vertices and/or removing small polygons that result from intersections of larger ones that border one another. A topological system also recognizes that boundaries to adjacent polygons are shared by the areas, so should not be ascribed attributes that belong to both areas (e.g., the shared border between two countries "belongs" to both countries).

By default, `fast()` will try to correct topological errors in vectors. There are three levels of correction, and they are not necessarily mutually exclusive:

1. **Automatic correction**: By default, `fast()` will apply automatic topology correction. You can turn this off using the `correct = FALSE` argument, though in most cases this is not recommended.
2. ***Manual* snapping and/or area removal**: In addition to correction from step 1, you can cause vertices of polygons close to one another to be "snapped" to same place and/or polygons that are smaller than some threshold to be removed. Problems with mis-aligned vertices arise when adjacent polygons are meant to share borders, but slight differences in the locations of the vertices cause them to mis-align. This mis-alignment can also produce small "slivers" of polygons that are the areas where they overlap. You can snap vertices within a given distance of one another using the `snap` argument followed by a numeric value, like `snap = 0.000001`. Units of `snap` are in map units (usually meters) for projected coordinate reference systems and degrees for unprojected systems (e.g., WGS84, NAD83, NAD27). You can also remove polygons that are smaller than a particular area using the `area` argument followed by a numeric value (e.g., `area = 1`). The units of `area` are in meters-squared, regardless of the coordinate reference system. Note that using `snap` and `area` entails some risk, as it is possible for nearby vertices to actually be distinct and for small areas to be legitimate.
3. **Automatic snapping and/or area removal**: In addition to the correction from step 1, you can use automatic `snap` and/or `area` correction on polygons vectors by setting `snap` and/or `area` to `NULL` (i.e., their default values). If just `snap` is `NULL`, only automatic snapping will be performed, and if just `area` is `NULL`, then only automatic area removal will be performed. Regardless, you will also need to set an integer value for `steps`, which is the number of steps to take between the smallest value of `snap` and/or `area` and the maximum value attempted. The function will then proceed by first attempting `snap = 0` and/or `area = 0` (i.e., no snapping or area removal). If this does not produce a topologically correct vector, **GRASS** will (internally)

suggest a range for `snap`. The `fast()` function then creates `steps` values from the lowest to the highest values of this range evenly-spaced along the log values of this range, then proceed to repeat the importing process until either the vector is imported correctly or the maximum value of `snap` is reached and results in a failed topology. Smaller values of `step` will result in more fine-grained attempts so are less likely to yield overcorrection, but can also take more time. The value of `area` in automatic correction is set to `snap^2`. **NB**: Automated snapping and area removal are only performed on polygons vectors, even if `snap` or `area` is `NULL`. To snap lines or points, you must set `snap` equal to a numeric value. The `area` correction is ignored for points and lines.

Issues can also arise due to:

- **Data table-vector mismatching**: If your vector has a data table ("attribute table") associated with it, errors can occur if there are more/fewer geometries (or multi-geometries) per row in the table. If you do not really need the data table to do your analysis, you can remove it (and thus obviate this error) using `dropTable = TRUE`.
- **Dissolving or aggregating "invalid" geometries**: Using the `resolve` argument, you can create a topologically valid vector by either coercing all overlapping portions of polygons into their own geometries (`resolve = "disaggregate"`), or by coercing them into a single, combined geometry (`resolve = "aggregate"`). Aggregation/disaggregation will be implemented after loading the vector into **GRASS** using the settings given by `snap` and `area`. Aggregation/disaggregation will cause any associated data table to be dropped (it forces `dropTable` to be `TRUE`). The default action is to do neither aggregation nor disaggregation (`resolve = NA`).

If none of these fixes work, you can try:

- **Correction outside of *fasterRaster***: Before you convert the vector into **fasterRaster**'s `GVector` format, you can also try using the `terra::makeValid()` or `sf::st_make_valid()` tools to fix issues, then use `fast()`.
- **Post-conversion to a `GVector`**: If you do get a vector loaded into `GVector` format, you can also use a set of **fasterRaster** vector-manipulation tools or `fillHoles()` to fix issues.

Value

A `GRaster` or `GVector`.

See Also

`rgrass::read_RAST()` and `rgrass::read_VECT()`, vector cleaning, `fillHoles()`, plus **GRASS** modules `v.in.ogr` (see `grassHelp("v.in.ogr")`) and `r.import` (see `grassHelp("r.import")`)

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Example data
madElev <- fastData("madElev") # integer raster
```

```
madCover <- fastData("madCover") # categorical raster
madCoast4 <- fastData("madCoast4") # polygons vector
madRivers <- fastData("madRivers") # lines vector
madDypsis <- fastData("madDypsis") # points vector

### Create GRasters from SpatRasters

# Create an integer raster:
elev <- fast(madElev)
elev

# Create a categorical raster:
cover <- fast(madCover)
madCover
levels(madCover) # category levels

# Create a GRaster from a file on disk:
rastFile <- system.file("extdata", "madForest2000.tif", package = "fasterRaster")
forest2000 <- fast(rastFile)
forest2000

# Create a 1's raster that spans the world:
ones <- fast(rastOrVect = "raster", crs = "epsg:4326")
ones

### Create GVectors

# Create a GVector from an sf vector:
coast4 <- fast(madCoast4)
coast4

# Create a GVector from a SpatVector:
madRivers <- vect(madRivers)
class(madRivers)
rivers <- fast(madRivers)
rivers

# Create a GVector from a vector on disk:
vectFile <- system.file("extdata/shapes", "madCoast.shp",
   package = "fasterRaster")
coast0 <- fast(vectFile)
coast0

# Import only Dypsis occurrences in a restricted area:
ant <- coast4[coast4$NAME_4 == "Antanambe"]
dypsisRestrict <- fast(madDypsis, extent = ant)
dypsis <- fast(madDypsis)

plot(coast4)
plot(ant, col = "gray80", add = TRUE)
plot(dypsis, add = TRUE)
plot(dypsisRestrict, col = "red", add = TRUE)
```

```
# Create a generic GVector that spans the world:
wallToWall <- fast(rastOrVect = "vector", crs = "epsg:4326") # WGS84
wallToWall

# Create a GVector from a numeric vector
pts <- c(-90.2, 38.6, -122.3, 37.9)
pts <- fast(pts, crs = "epsg:4326") # WGS84

# Create a GVector from a matrix (can also use data.frame or data.table):
mat <- matrix(c(-90.2, 38.6, -122.3, 37.9), ncol = 2, byrow = TRUE)
mat <- fast(mat, crs = "epsg:4326", keepgeom = TRUE) # WGS84

}
```

`fastData` *Get one of the example rasters or spatial vectors*

Description

This function is a simple way to get example rasters or spatial vector datasets that come with **fasterRaster**.

Usage

```
fastData(x)
```

Arguments

`x`	The name of the raster or spatial vector to get. All of these represent a portion of the eastern coast of Madagascar. Spatial vectors (objects of class `sf` from the **sf** package): • madCoast0: Outline of the region (polygon) • madCoast4: Outlines of the Fokontanies (Communes) of the region (polygons) • madDypsis: Records of plants of the genus *Dypsis* (points) • madRivers: Major rivers (lines) Rasters (objects of class `SpatRaster` from the **terra** package, saved as GeoTIFF files): • madChelsa: Bioclimatic variables • madCover: Land cover • `madElev`: Elevation • madForest2000: Forest cover in year 2000 • madForest2014: Forest cover in year 2014 • madLANDSAT: Surface reflectance in 2023 • madPpt, madTmin, madTmax: Rasters of mean monthly precipitation, and minimum and maximum temperature.

Data frames

- appFunsTable: Table of functions usable by app()
- madCoverCats: Land cover values and categories for madCover
- vegIndices: Vegetation indices that can be calculated with vegIndex()

Value

A SpatRaster, sf spatial vector, or a data.frame.

Examples

```
### vector data

library(sf)

# For vector data, we can use data(*) or fastData(*):
data(madCoast0) # same as next line
madCoast0 <- fastData("madCoast0") # same as previous
madCoast0
plot(st_geometry(madCoast0))

madCoast4 <- fastData("madCoast4")
madCoast4
plot(st_geometry(madCoast4), add = TRUE)

madRivers <- fastData("madRivers")
madRivers
plot(st_geometry(madRivers), col = "blue", add = TRUE)

madDypsis <- fastData("madDypsis")
madDypsis
plot(st_geometry(madDypsis), col = "red", add = TRUE)

### raster data

library(terra)

# For raster data, we can get the file directly or using fastData(*):
rastFile <- system.file("extdata/madElev.tif", package="fasterRaster")
madElev <- terra::rast(rastFile)

madElev <- fastData("madElev") # same as previous two lines
madElev
plot(madElev)

madForest2000 <- fastData("madForest2000")
madForest2000
plot(madForest2000)

madForest2014 <- fastData("madForest2014")
madForest2014
plot(madForest2014)
```

```
# multi-layer rasters
madChelsa <- fastData("madChelsa")
madChelsa
plot(madChelsa)

madPpt <- fastData("madPpt")
madTmin <- fastData("madTmin")
madTmax <- fastData("madTmax")
madPpt
madTmin
madTmax


# RGB raster
madLANDSAT <- fastData("madLANDSAT")
madLANDSAT
plotRGB(madLANDSAT, 4, 1, 2, stretch = "lin")

# categorical raster
madCover <- fastData("madCover")
madCover
madCover <- droplevels(madCover)
levels(madCover) # levels in the raster
nlevels(madCover) # number of categories
catNames(madCover) # names of categories table

plot(madCover)
```

`faster` *Set or get options shared across fasterRaster functions*

Description

`faster()` either sets or gets options used across **fasterRaster** functions. Its use can vary:

- Get current values of a particular option: Use `faster("option_name")`. Values will remain unchanged.
- Get current values of all options: Use `faster()` (no arguments). Values will remain unchanged.
- Get default values of a particular option: Use `faster("option_name", default = TRUE)`. Values will remain unchanged.
- Get default values of all options: Use `faster(default = TRUE)`. Values will remain unchanged.
- Set values of particular options: Use the form `faster(option 1 = value1, option2 = value2)`.
- Set all options to their defaults: Use `faster(restore = TRUE)`.

You cannot simultaneously get and set options.

To run most **fasterRaster** functions, you must set the `grassDir` option.

Usage

```
faster(..., default = FALSE, restore = FALSE)
```

Arguments

...	Either: • A character vector: Name(s) of option(s) to get values of; • An option and the value of the option using an `option = value` pattern; or • A names `list` with names that match the options you wish to change, and with values to assign to each option. Options include: • `grassDir` (character): The folder in which **GRASS** is installed on your computer. You must set this option to run most **fasterRaster** functions. Depending on your operating system, your install directory will look something like this: – Windows: `"C:/Program Files/GRASS GIS 8.4"` – Mac OS: `"/Applications/GRASS-8.4.app/Contents/Resources"` – Linux: `"/usr/local/grass"` • `addonsDir` (character): Folder in which **GRASS** addons are stored. If `NA` and `grassDir` is not `NA`, this will be assumed to be `file.path(grassDir, "addons")`. The default values is `NA`. • `cores` (integer/numeric integer): Number of processor cores to use on a task. The default is 2. Some **GRASS** modules are parallelized. • `memory` (integer/numeric): The amount of memory to allocate to a task, in GB, for **GRASS**. The default is 2048 MB (i.e., 2 GB). Some **GRASS** modules can take advantage of more memory. • `useDataTable` (logical): If `FALSE` (default), functions that return tabular output produce `data.frames`. If `TRUE`, output will be `data.tables` from the **data.table** package. This can be much faster, but it might require you to know how to use `data.tables` if you want to manipulate them in **R**. You can always convert them to `data.frames` using `base::as.data.frame()`. • `verbose` (logical): If `TRUE`, show progress during function operations and other messages. Default is `FALSE`. This overrides the value of any `verbose` argument in a function. • `debug` (logical): If `TRUE`, show **GRASS** messages and otherwise hidden slots in classes. This is mainly used for debugging, so most users will want to keep this at its default, `FALSE`. • `workDir` (character): The folder in which **GRASS** rasters, vectors, and other objects are created and manipulated. By default, this is given by `tempdir()`. Note that on some systems, changing the default folder to somewhere else can cause problems with **fasterRaster** being able to find rasters in **GRASS** that have been created.
`default`	Logical: Return the default value(s) of the option(s). The default value of `default` is `FALSE`.
`restore`	Logical: If `TRUE`, the all options will be reset to their default values. The default is `FALSE`.

Value

If options are changed, then a named list of option values *before* they were changed is returned invisibly.

If option values are requested, a named list with option values is returned (not invisibly).

Examples

```
if (grassStarted()) {

# See current values for options:
faster("grassDir")
faster("cores")
faster("memory")
faster("useDataTable")
faster() # all options

# See default values for options:
faster("cores", default = TRUE)
faster(default = TRUE) # all options

# Set options (change accordingly for your system!!!)
if (FALSE) {

   opts. <- faster() # remember starting values of options

   faster(grassDir = "C:/Program Files/GRASS GIS 8.4")
   faster(verbose = TRUE, memory = 1024, cores = 1)

   faster(c("grassDir", "verbose", "memory", "cores"))

   faster(opts.) # reset options to starting values

}

}
```

---

fillHoles,GVector-method

Fill holes in a GVector

Description

`fillHoles()` removes holes in a `GVector`.

Usage

```
## S4 method for signature 'GVector'
fillHoles(x, fail = TRUE)
```

Arguments

x	A GVector.
fail	Logical: If TRUE (default), and **GRASS 8.3** or higher is not installed, cause an error. If FALSE, a warning will be displayed and a NULL value will be returned. This function requires **GRASS 8.3** or higher to be installed.

Value

A GVector.

See Also

terra::fillHoles(), **GRASS** manual page for module v.fill.holes (see grassHelp("v.fill.holes"))

Examples

```
if (grassStarted()) {

# Setup
library(sf)

# Example data:
madCoast4 <- fastData("madCoast4")
madRivers <- fastData("madRivers")
madDypsis <- fastData("madDypsis")

# Convert sf vectors to GVectors:
coast <- fast(madCoast4)
rivers <- fast(madRivers)
dypsis <- fast(madDypsis)

# Geographic properties:
ext(rivers) # extent
crs(rivers) # coordinate reference system
st_crs(rivers) # coordinate reference system
coordRef(rivers) # coordinate reference system

# Column names and data types:
names(coast)
datatype(coast)

# Points, lines, or polygons?
geomtype(dypsis)
geomtype(rivers)
geomtype(coast)

is.points(dypsis)
is.points(coast)

is.lines(rivers)
is.lines(dypsis)
```

```
is.polygons(coast)
is.polygons(dypsis)

# Number of dimensions:
topology(rivers)
is.2d(rivers) # 2-dimensional?
is.3d(rivers) # 3-dimensional?

# Just the data table:
as.data.frame(rivers)
as.data.table(rivers)

# Top/bottom of the data table:
head(rivers)
tail(rivers)

# Vector or table with just selected columns:
names(rivers)
rivers$NAME
rivers[[c("NAM", "NAME_0")]]
rivers[[c(3, 5)]]

# Select geometries/rows of the vector:
nrow(rivers)
selected <- rivers[2:6]
nrow(selected)

# Plot:
plot(coast)
plot(rivers, col = "blue", add = TRUE)
plot(selected, col = "red", lwd = 2, add = TRUE)

# Vector math:
hull <- convHull(dypsis)

un <- union(coast, hull)
sameAsUnion <- coast + hull
plot(un)
plot(sameAsUnion)

inter <- intersect(coast, hull)
sameAsIntersect <- coast * hull
plot(inter)
plot(sameAsIntersect)

er <- erase(coast, hull)
sameAsErase <- coast - hull
plot(er)
plot(sameAsErase)

xr <- xor(coast, hull)
sameAsXor <- coast / hull
```

```
plot(xr)
plot(sameAsXor)

# Vector area and length:
expanse(coast, unit = "km") # polygons areas
expanse(rivers, unit = "km") # river lengths

### Fill holes

# First, we will make some holes by creating buffers around points.
buffs <- buffer(dypsis, 500)

holes <- coast - buffs
plot(holes)

filled <- fillHoles(holes, fail = FALSE)

}
```

`fillNAs,GRaster-method`

Fill NA cells in a raster using interpolation

Description

This function uses splines to fill `NA` cells in a raster based on the values of nearby cells. Depending on the method used, not all `NA` cells can be filled.

Usage

```
## S4 method for signature 'GRaster'
fillNAs(
  x,
  lambda = NULL,
  method = "bilinear",
  min = -Inf,
  max = Inf,
  cells = Inf
)
```

Arguments

`x`	A `GRaster`.
`lambda`	Either `NULL` (default), or a numeric value > 0: If `NULL`, then the function will use leave-one-out crossvalidation to find the optimal value.
`method`	Character: Type of spline, either `"bilinear"` (default), `"bicubic"`, or `"RST"` (regularized splines with tension). Partial matching is used and case is ignored. **Note**: The RST method will often display warnings, but these can be ignored.

`min`, `max` Numeric: Lowest and highest values allowed in the interpolated values. Values outside these bounds will be truncated to the minimum/maximum value(s) allowed. The default imposes no constraints. For multi-layered rasters, you can supply a single value for `min` and/or `max`, or multiple values (one per layer). Values will be recycled if there are fewer than one or them per layer in the raster.

`cells` Integer or numeric integer: Number of cells away from the non-NA cells to fill. For example, if `cells = 2`, then only cells within a 2-cell buffer of non-NA cells will be filled. The default is `Inf` (fill all possible cells–some methods may not be able to do this, depending on the configuration of the raster).

Value

A `GRaster`.

See Also

`terra::interpNear()`, **GRASS** module `r.fillnulls` (see `grassHelp("r.fillnulls")`)

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Elevation raster:
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster:
elev <- fast(madElev)

### Fill NAs:
bilinear <- fillNAs(elev)
bicubic <- fillNAs(elev, method = "bicubic")
rst <- fillNAs(elev, method = "rst")

maps <- c(elev, bilinear, bicubic, rst)
names(maps) <- c("original", "bilinear", "bicubic", "RST")
plot(maps)

### Constrain interpolated values to > 0
constrained <- fillNAs(elev, min = 0)

# Compare unconstrained and constrained:
minmax(bilinear)
minmax(constrained)

### Interpolate to only first 10 cells away from non-NA cells:
restrained <- fillNAs(elev, cells = 10)

maps <- c(elev, restrained)
```

```
names(maps) <- c("Original", "within 10 cells")
plot(maps)

}
```

`flow,GRaster-method` *Identify watershed basins and direction and accumulation of flow*

Description

The `flow()` function uses a raster representing elevation to compute other rasters representing:

- Flow accumulation;
- Direction of flow;
- Watershed basins;
- Flooded areas; and/or
- Topographic convergence (log of flow accumulation divided by local slope).

More details about the computations can be found at the help page for the **GRASS** module `r.terraflow`] (see `grassHelp("r.terraflow")`)

Usage

```
## S4 method for signature 'GRaster'
flow(
  x,
  direction = "multi",
  return = "accumulation",
  dirThreshold = Inf,
  scratchDir = NULL
)
```

Arguments

`x`	A `GRaster` with a single layer, typically representing elevation.
`direction`	Character: Either `"single"` or `"multi"`. This indicates whether a single-direction flow or multi-direction flow model is used. The default is `"multi"`. Partial matching is used and case is ignored.
`return`	Character vector: Indicates what rasters to return. Partial matching is used and case is ignored. Options include: • `"accumulation"` (default): Flow accumulation raster. • `"basins"`: Watershed basins • `"direction"`: Flow direction • `"flooded"`: Flooded areas • `"TCI"`: Topographic convergence index

- "*": All of the above

dirThreshold — Numeric (default is Inf): For the multi-direction flow model, this indicates the amount of accumulated flow above which the single-direction flow rule is used to locate the egress of water from a cell. This is the d8cut parameter in r.stream.extract.

scratchDir — Character or NULL (default): Directory in which to store temporary files. The **GRASS** module r.terraflow makes a lot of temporary files. If this is NULL, then a temporary folder in the user's working directory will be used (see getwd()).

Value

A GRaster.

See Also

flowPath(), streams(), the **GRASS** module r.terraflow (see grassHelp("r.terraflow"))

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")
elev <- fast(madElev)

# Calculate flow accumulation and watershed basins
water <- flow(elev, return = c("accum", "basins"))
water

elevWater <- c(elev, water)
plot(elevWater)

}
```

---

flowPath,GRaster-method

Path of water flow across a landscape

Description

This function finds the least-cost pathway from a set of starting points to the lowest cells accessible from them while, in each step, traversing "down" slope gradients. It is intended to depict the path a drop of water would take when flowing across a landscape. For a single starting point, the defaults settings will produce a raster with cells with values of 1 along the path. All other cells will be set to NA.

Usage

```
## S4 method for signature 'GRaster'
flowPath(x, y, return = "ID")
```

Arguments

x	A GRaster with a single layer, typically representing elevation.
y	A "points" GVector. The GVector must have <= 1024 points.
return	Character: Indicates the type of values "burned" into the cells of the output raster. Case is ignored and partial matching is used, but only one option can be selected. • "ID" (default): Cells in each path are labeled with the index of the starting point. A cell in the flow path of the first point will have a value of 1, a cell in the flow path of the second point will have a value of 2, and so on. • "sequence": The output raster's cells will start with 1 at the source point(s), then accumulate so that the next cell in the flow path is 2, the one after that 3, and so on. • "copy": The cells in the flow path will have the elevation raster's values in the cells along the flow path(s). • "accumulation": Cells in the flow path will accumulate the elevation raster's cell values. For example, if the starting cell has an elevation of 700 and the next cell in the drainage path has a value of 600 and the one after that 500, then the first cell in the path will have a value of 700, the next 1300 (= 700 + 600), and the third 1800 (= 700 + 600 + 500). • A numeric value: All cells in flow paths will be assigned this value.

Value

A GRaster.

See Also

flow(), streams(), the **GRASS** module r.drain (see grassHelp("r.drain"))

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")
madCoast4 <- fastData("madCoast4")

# Convert to GRaster and crop to a sub-portion (easier for visualizing)
elev <- fast(madElev)
coast4 <- fast(madCoast4)
```

```
ant <- coast4[coast4$NAME_4 == "Antanambe"]
elevAnt <- crop(elev, ant)

# Create a set of random points to serve as starting points:
starts <- spatSample(elevAnt, 10, as.points = TRUE, seed = 2)

# Remove points in water:
starts <- starts[complete.cases(starts)]

# Calculate flow paths and label each by ID:
paths <- flowPath(elevAnt, starts)
paths

plot(elevAnt, legend = FALSE, main = "Flow path for each point")
plot(paths, add = TRUE)
plot(starts, pch = 1, add = TRUE)

# Flow paths with cell values indicating number of cells from each start:
seqs <- flowPath(elevAnt, starts, return = "seq")

plot(elevAnt, legend = FALSE, main = "Sequentially-numbered flow paths")
plot(seqs, add = TRUE)
plot(starts, pch = 1, add = TRUE)

# We can convert flow paths to lines:
seqLines <- as.lines(seqs)
plot(seqLines)
seqLines

}
```

`focal,GRaster-method` *Calculate cell values based on values of nearby cells*

Description

This function calculates statistics on a moving "neighborhood" of cells of a raster. The neighborhood can be a square, circle, or a user-defined set of cells (with or without weights).

Usage

```
## S4 method for signature 'GRaster'
focal(x, w = 3, fun = "sum", circle = FALSE, quantile = 0.5)
```

Arguments

`x`	A `GRaster`.
`w`	Numeric integer or a square matrix with an odd number of rows and columns: The size and nature of the neighborhood:

- "Square" neighborhoods (when `circle = FALSE`): An odd integer >= 3, indicating indicates the size of a "square" neighborhood (number of cells wide and number or cells tall).
- "Circular" neighborhoods (when `circle = TRUE`): An odd integer >=3, indicating the diameter of the circle.
- A matrix of cell weights: The matrix must be square and have an odd number of rows and columns (example: `matrix(c(0.5, 1, 0.5, 1, 2, 1, 0.5, 1, 0.5), nrow=3)`). You cannot use a weights matrix when `circle = TRUE`. Cells with `NA` as a weight will be ignored. Note that weighted matrices should not be used for function `min`, `max`, `count`, `nunique`, or `interspersion`.

`fun` Character: Name of the function to apply to the neighborhood:

- "`mean`" (default)
- "`median`"
- "`mode`"
- "`min`" or "`max`": Minimum or maximum. Should not use a weights matrix.
- "`range`": Difference between the maximum and minimum. Should not use a weights matrix.
- "`sd`": Sample standard deviation. NB: This is the same as the `stats::sd()` function.
- "`sdpop`": Population standard deviation. NB: This is the same as the function "stddev" in the **GRASS** module `r.neighbors`.
- "`sum`": Sum of non-'NA' cells.
- "`count`": Number of non-'NA cells. Should not use a weights matrix.
- "`var`": Sample variance. NB: This is the same as the `stats::var()` function.
- "`varpop`": Population variance. NB: This is the same as the function "variance" in the **GRASS** module `r.neighbors`.
- "`nunique`": Number of unique values. Should not use a weights matrix.
- "`interspersion`": Proportion of cells with values different from focal cell (e.g., if 6 of 8 cells have different values, then the interspersion is 6/8 = 0.75). NB: This is slightly different from how it is defined in the **GRASS** module `r.neighbors`. Should not use a weights matrix.
- "`quantile`": Quantile of values. The value in argument `quantile` is used to specify the quantile.

The center cell value is always included in the calculations, and all calculations ignore `NA` cells (i.e., they do not count as cells in the focal neighborhood).

`circle` Logical: If `FALSE` (default), use a square neighborhood. If `TRUE`, use a circular neighborhood. When this is `TRUE`, argument `w` cannot be a matrix.

`quantile` Numeric between 0 and 1, inclusive: Quantile to calculate when `fun = "quantile"`. The default value is 0.5 (median), and valid values must be in the range between 0 and 1, inclusive.

Value

A `GRaster`.

See Also

terra::focal(), **GRASS** manual page for module `r.neighbors` (see `grassHelp("r.neighbors")`)

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster:
elev <- fast(madElev)

# Focal calculations:
sums <- focal(elev, fun = "sum")
means <- focal(elev, fun = "mean")

# Focal calculations on a circular window:
sds <- focal(elev, fun = "sd") # square
sdsCircle <- focal(elev, fun = "sd", circle = TRUE) # circle

sds
sdsCircle

plot(sds - sdsCircle)

# Focal calculations with user-defined weights:
w <- matrix(c(1, 0, 1, 0, 1, 0, 1, 0, 1), ncol = 3)
w
sumsWeighted <- focal(elev, fun = "sum", w = w)

s <- c(sums, sumsWeighted)
minmax(s)

}
```

---

fractalRast,GRaster-method

Create fractal raster

Description

`fractalRast()` creates a raster with a fractal pattern.

Usage

```
## S4 method for signature 'GRaster'
fractalRast(x, n = 1, mu = 0, sigma = 1, dimension = 2.05)
```

Arguments

x	A GRaster. The output will have the same extent and dimensions as this raster.
n	A numeric integer: Number of rasters to generate.
mu, sigma	Numeric: Mean and sample standard deviation of output.
dimension	Numeric: Fractal dimension. Must be between 2 and 3.

Value

A GRaster.

See Also

rSpatialDepRast(), rnormRast(), runifRast(), **GRASS** manual page for module r.surf.fractal (see grassHelp("r.surf.fractal"))

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Elevation raster
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster:
elev <- fast(madElev)

### Create a raster with values drawn from a uniform distribution:
unif <- runifRast(elev)
plot(unif)

### Create a raster with values drawn from a normal distribution:
norms <- rnormRast(elev, n = 2, mu = c(5, 10), sigma = c(2, 1))
plot(norms)
hist(norms, bins = 100)

# Create a raster with random, seemingly normally-distributed values:
rand <- rSpatialDepRast(elev, dist = 1000)
plot(rand)

# Values appear normal on first inspection:
hist(rand)

# ... but actually are patterned:
hist(rand, bins = 100)

# Create a fractal raster:
fractal <- fractalRast(elev, n = 2, dimension = c(2.1, 2.8))
```

```
plot(fractal)
hist(fractal)

}
```

`fragmentation,SpatRaster-method`
Landscape fragmentation class following Riitters et al. (2020)

Description

Riitters et al. (2020) propose a classification scheme for forest fragmentation (which can be applied to any habitat type). The scheme relies on calculating density (e.g., number of forested cells in a window around a focal cell) and connectivity (number of cases where neighboring cells are both forested). This function calculates these classes from a `GRaster` or `SpatRaster` in which the focal habitat type has cell values of 1, and non-focal habitat type has cell values of 0 or `NA`.

Note that by default, the `SpatRaster` and `GRaster` versions will create different results around the border of the raster. The `SpatRaster` version uses the `terra::focal()` function, which will not return an `NA` value when its window overlaps the raster border if the `na.rm` argument is `TRUE`. However, the `GRaster` version uses the **GRASS** module `r.neighbors`, which does return `NA` values in these cases.

The fragmentation classes are:

- Value provided by `none`: None (i.e., no forest; default is `NA`).
- 1: Patch
- 2: Transitional
- 3: Perforated
- 4: Edge
- 5: Undetermined (not possible to obtain when `w = 3`)
- 6: Interior

Usage

```
## S4 method for signature 'SpatRaster'
fragmentation(
  x,
  w = 3,
  undet = "undetermined",
  none = NA,
  na.rm = TRUE,
  cores = faster("cores"),
  verbose = TRUE
)

## S4 method for signature 'GRaster'
fragmentation(x, w = 3, undet = "undetermined", none = NA, verbose = TRUE)
```

Arguments

x	A SpatRaster or GRaster.
w	An odd, positive integer: Size of the window across which fragmentation is calculated (in units of "rows" and "columns"). The default is 3, meaning the function uses a 3x3 moving window to calculate fragmentation. For large rasters, compute time is ~O(N) + O(N * w^2), where N is the number of cells in the raster. So, even a small increase in w can increase compute time by a lot.
undet	Character: How to assign the "undetermined" case. Valid values are "perforated" (default), "edge", and "undetermined". Partial matching is used. If Pf is the proportional density raster cell value and Pff the proportional connectivity raster cell value, the undetermined case occurs when Pf > 0.6 and Pf == Pff.
none	Integer or NA (default): Value to assign to a cell with no focal habitat. Riitters et al. use NA. This will be forced to an integer if it is not an actual integer.
na.rm	Logical: If TRUE (default) and x is a SpatRaster, then cells near the edge of the raster where the window overlaps the edge can still be assigned a fragmentation class. If FALSE, these cells will be assigned a value of none.
cores	Integer: Number of processor cores to use for when processing a SpatRaster.
verbose	Logical: If TRUE (default), display progress.

Value

A categorical SpatRaster or GRaster. The values assigned to each class can be seen with levels().

References

Riitters, K., J. Wickham, R. O'Neill, B. Jones, and E. Smith. 2000. Global-scale patterns of forest fragmentation. *Conservation Ecology* 4:3. URL: http://www.consecol.org/vol4/iss2/art3/. Also note the errata.

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data:
madForest <- fastData("madForest2000") # raster

### Fragmentation classes from a SpatRaster

fragTerra <- fragmentation(madForest)
plot(fragTerra)
levels(fragTerra)
freq(fragTerra)

### Fragmentation classes from a GRaster
```

```
# Convert to GRaster:
forest <- fast(madForest)

# Fragmentation class:
frag <- fragmentation(forest)
plot(frag)
levels(frag)
freq(frag)

}
```

`freq,GRaster-method` *Frequencies of cell values in a raster*

Description

`freq()` tabulates the frequency of cell values in a raster. For rasters where `datatype()` is `integer` or `factor`, the frequency of each value or level is reported. For other rasters, the range of values is divided into bins, and the number of cells with values in each bin is reported.

Usage

```
## S4 method for signature 'GRaster'
freq(x, digits = 3, bins = 100, value = NULL)
```

Arguments

`x`	A `GRaster`.
`digits`	Numeric integer: Number of digits by which to round raster values. Ignored for integer and categorical rasters.
`bins`	Positive numeric integer: Number of bins in which to divide values of `numeric` rasters. The default is 100. For `integer` and categorical rasters, each value is tallied (i.e., this is ignored).
`value`	Numeric or `NULL` (default): If numeric, only cells with this value will be counted. If `NULL`, all values will be counted.

Value

A `data.frame` or a named `list` of `data.frames`, one per layer in `x`.

See Also

`terra::freq()`, module `r.stats` in **GRASS**

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev") # raster
madCover <- fastData("madCover") # categorical raster

# Convert to GRasters
elev <- fast(madElev) # raster
cover <- fast(madCover) # categorical raster

# Frequencies of integer raster values
f <- freq(elev)
print(f) # have to do this sometimes if output is a data table

# Frequencies of categorical raster values
f <- freq(cover)
print(f) # have to do this sometimes if output is a data table

# Frequencies of given values
f <- freq(elev, value = 1)
print(f) # have to do this sometimes if output is a data table

# When a GRaster has non-integer values, they will be binned:
f <- freq(elev + 0.1, bins = 10)
print(f)

}
```

`geomorphons,GRaster-method`

Identify terrain feature types

Description

Geomorphons are idealized terrain types calculated from an elevator raster based on a moving window of a given size. The window is a torus (which can have an inner radius of 0, so can also be a circle), which allows it to identify geomorphons of a given size while ignoring ones larger or smaller. There are 10 basic geomorphons. Consult the the manual for **GRASS** module `r.geomorphon` using `grassHelp("r.geomorphon")` for more details and diagrams of each type of geomorphon. Geomorphon types include:

1. Flat areas: Focal area has approximately the same elevation as surrounding areas
2. Pits: An area is lower than all other surrounding areas
3. Valley: Focal area has elevation similar to two opposing side of the window but lower than the other two opposing sides

4. Footslope: Focal region is at the "bottom" of a slope
5. Hollow: A small valley/indention in the crest of a hill
6. Slope: Cells in the window form an approximately uniform slope
7. Spur: An extrusion at the foot of a hill (i.e.,, a small hill extending out from the foot of a slope)
8. Shoulder: The crest of a slope
9. Ridge: Opposite of a valley; focal area is higher than two opposing sides but approximately the same elevation as the other two opposing sides
10. Peak: Focal area is higher than any other in the window

Usage

```
## S4 method for signature 'GRaster'
geomorphons(
  x,
  inner = 0,
  outer = 3,
  unit = "cells",
  flat = 1,
  flatDist = 0,
  mode = "1"
)
```

Arguments

x	A single-layer GRaster, typically representing elevation.
inner, outer	Integer: Inner and outer radii of the torus used to identify geomorphons, in cells or meters (set by argument unit). The inner default value is 0 and the outer default value is 3. The outer radius sets the maximum size of a geomorphon that that can be identified, and inner sets the smallest size. If unit is "meters", the value of outer must be larger than the smaller dimension of any cell in the east-west and north-south directions.
unit	Character: Units of inner and outer; can be either "cells" (default) or "meters". Partial matching is used.
flat	Numeric value >= 0: Minimum difference (in degrees) between the focal area areas around it for a geomorphon to be considered as "flat". Larger cells (i.e., ~1 km resolution or larger) require smaller values («1) to correctly identify flat areas. Higher values result in more areas being classified as "flat" geomorphons. The default value is 1.
flatDist	Numeric: Distance (in meters) to correct for the effect of large distances on the diminished capacity to identify "flat" geomorphons. If the distance between the focal area and a surrounding area surpasses this distance, then the effective value of flat will be reduced
mode	Character: Method for implementing the zenith/line-of-site search. Partial matching is used: • "1" (default): The "original" geomorphon mode (in **GRASS** module r.geomorphon, the "anglev1" method)

- "2": Better handling of cases with equal zenith/nadir angles (the "anglev2" method)
- "2d": As "2", but takes into account zenith/nadir distance ("anglev2_distance" method)

Value

A categorical GRaster where each geomorphon is a category (see vignette("GRasters", package = "fasterRaster")).

See Also

GRASS manual for module r.geomorphon (see grassHelp("r.geomorphon"))

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster:
elev <- fast(madElev)

# Geomorphons:
geos <- geomorphons(elev)
geos
levels(geos) # levels
freq(geos) # frequencies across cells

col <- c("gray90", "red", "orange", "blue", "green", "pink", "firebrick",
  "purple", "gray50", "black")
plot(geos, col = col)

}
```

---

geomtype,GVector-method

Geometry of a GVector (points, lines, or polygons)

Description

geomtype() reports whether a GVector represents points, lines, or polygons. The "is.*" functions test whether the GVector represents points, lines, or polygons.

Usage

```
## S4 method for signature 'GVector'
geomtype(x, grass = FALSE)

## S4 method for signature 'GVector'
is.points(x)

## S4 method for signature 'GVector'
is.lines(x)

## S4 method for signature 'GVector'
is.polygons(x)
```

Arguments

x	A GVector.
grass	Logical: If FALSE (default), return **terra**-like geometry types ("points", "lines", or "polygons"). If TRUE, return **GRASS**-like geometry types ("point", "line", "area"–note that these are a subset of the available types and may not be the "true" **GRASS** type).

Value

geomtype() returns either "points", "lines", or "polygons" if the grass arguments is FALSE, or "point", "line", "area" if grass is TRUE. The "is.*" functions return TRUE or FALSE.

See Also

terra::geomtype()

Examples

```
if (grassStarted()) {

# Setup
library(sf)

# Example data:
madCoast4 <- fastData("madCoast4")
madRivers <- fastData("madRivers")
madDypsis <- fastData("madDypsis")

# Convert sf vectors to GVectors:
coast <- fast(madCoast4)
rivers <- fast(madRivers)
dypsis <- fast(madDypsis)

# Geographic properties:
ext(rivers) # extent
crs(rivers) # coordinate reference system
```

```
st_crs(rivers) # coordinate reference system
coordRef(rivers) # coordinate reference system

# Column names and data types:
names(coast)
datatype(coast)

# Points, lines, or polygons?
geomtype(dypsis)
geomtype(rivers)
geomtype(coast)

is.points(dypsis)
is.points(coast)

is.lines(rivers)
is.lines(dypsis)

is.polygons(coast)
is.polygons(dypsis)

# Number of dimensions:
topology(rivers)
is.2d(rivers) # 2-dimensional?
is.3d(rivers) # 3-dimensional?

# Just the data table:
as.data.frame(rivers)
as.data.table(rivers)

# Top/bottom of the data table:
head(rivers)
tail(rivers)

# Vector or table with just selected columns:
names(rivers)
rivers$NAME
rivers[[c("NAM", "NAME_0")]]
rivers[[c(3, 5)]]

# Select geometries/rows of the vector:
nrow(rivers)
selected <- rivers[2:6]
nrow(selected)

# Plot:
plot(coast)
plot(rivers, col = "blue", add = TRUE)
plot(selected, col = "red", lwd = 2, add = TRUE)

# Vector math:
hull <- convHull(dypsis)
```

```
un <- union(coast, hull)
sameAsUnion <- coast + hull
plot(un)
plot(sameAsUnion)

inter <- intersect(coast, hull)
sameAsIntersect <- coast * hull
plot(inter)
plot(sameAsIntersect)

er <- erase(coast, hull)
sameAsErase <- coast - hull
plot(er)
plot(sameAsErase)

xr <- xor(coast, hull)
sameAsXor <- coast / hull
plot(xr)
plot(sameAsXor)

# Vector area and length:
expanse(coast, unit = "km") # polygons areas
expanse(rivers, unit = "km") # river lengths

### Fill holes

# First, we will make some holes by creating buffers around points.
buffs <- buffer(dypsis, 500)

holes <- coast - buffs
plot(holes)

filled <- fillHoles(holes, fail = FALSE)

}
```

`global,GRaster-method` *Summary statistics for GRasters*

Description

`global()` calculates a summary statistic across all the cells of a `GRaster`. It returns a single value for each layer of the raster.

Usage

```
## S4 method for signature 'GRaster'
global(x, fun = "mean", probs = seq(0, 1, 0.25), ...)

## S4 method for signature 'missing'
global(x, ...)
```

Arguments

x	A GRaster or missing. If missing, then a vector of all of the accepted function names is returned.
fun	Character vector: The name of the function(s): • "*": All of the functions below. • "cv": Sample coefficient of variation (expressed as a proportion of the mean). • "cvpop": Population coefficient of variation (expressed as a proportion of the mean). • "max" and "min": Highest and lowest values across non-NA cells. NB: minmax() is faster. • "mean" (default): Average. • "meanAbs": Mean of absolute values. • "median": Median. • "quantile": Quantile (see also argument probs). • "range": Range. Note that following terra::global(), the minimum and maximum are reported, not the actual range. • "sd": Sample standard deviation (same as stats::sd()). • "sdpop": Population standard deviation. • "sum": Sum. • "var": Sample variance (same as stats::var()). • "varpop": Population variance.
probs	Numeric within the range from 0 to 1: Quantile(s) at which to calculate quantile.
...	Other arguments (unused).

Value

If x is missing, the function returns a character vector of all accepted function names. If x is a GRaster, a data frame with the specified statistics is returned.

See Also

terra::global() and **GRASS** module r.univar

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster:
elev <- fast(madElev)
```

```
# What functions can we use with global()?
global()

# Calculate global statistics:
global(elev, fun = c("mean", "var", "varpop"))
global(elev, "quantile", probs = c(0.25, 0.5, 0.75))

global(elev, "*") # calculate all available functions

}
```

GLocation-class *Classes for fasterRaster sessions, regions, rasters, and vectors*

Description

The G-suite of S4 classes contain pointers to **GRASS** objects or metadata about the current **GRASS** session. Most users will manipulate objects using these classes, but do not need to know the details.

- The GLocation class stores information about the **GRASS** "project"/"location"(see `vignette("projects_mapsets", package = "fasterRaster")`), and coordinate reference system. Contained by all the rest.
- The GSpatial class contains the GLocation class and stores information about spatial objects (extent, topology) plus the name of the file representing it in **GRASS** (its source). Contained by GRegion, GRaster, and GVector.
- The GRegion class contains the GSpatial class and stores information about grids (dimensions and resolution). They do have sources, but these are not used (they're always NA). Contained by GRaster. The GRegion corresponds to **GRASS** "regions", though GRegion objects are not actually pointers to **GRASS** "region" files (see `vignette("regions", package = "fasterRaster")`).
- The GRaster class contains the GRegion class and represents rasters. It stores information on number of layers, categories, min/max values, and user-friendly names. Categorical GRasters are associated with a "levels" table for representing categorical data (e.g., wetlands, forest, etc.).
- The GVector class contains the GSpatial class and represents spatial vectors. It may or may not have an associated data.table (i.e., a data.frame), which contains metadata about each geometry in the vector.

Value

An object of class GLocation, GSpatial, GRegion, GRaster, or GVector.

Slots

location Character (all classes): The **GRASS** "project"/"location" of the object. The default value is default. Can be obtained using the hidden function .location(). See `vignette("projects_mapsets", package = "fasterRaster")`.

`mapset` Character (all classes): The **GRASS** "mapset". Default value is `PERMANENT`. Typically hidden to users. Can be obtained using the hidden function `.mapset()`. See `vignette("projects_mapsets", package = "fasterRaster")`.

`workDir` Character (all classes): Directory in which **GRASS** stores files.

`topology` Character (`GSpatial` objects, including `GRegions`, `GRasters`, and `GVectors`): Valid values are `2D` (2-dimensional–most rasters and vectors) or `3D` (3-dimensional–e.g., LIDAR data). Can be obtained using `topology()`.

`sources` Character (`GRasters` and `GVectors`): Name of the object in **GRASS**. These are typically made on-the-fly and provide the pointer to the object from **R** to **GRASS**. Changing them manually will break the connection. Can be obtained using `sources()`.

`names` Character (`GRasters` only): Name of a raster or each raster layer in. Can be obtained using `names()`.

`crs` Character (all classes): Coordinate reference systems string (preferably in WKT2 format). Can be obtained using `crs()` or `st_crs()`.

`projection` Character: The **GRASS** "projection" for a `GRaster` or `GVector`. Can be obtained using `.projection()`.

`dimensions` Dimensions:

- `GRegions` and `GRasters`: Vector of three integers indicating number of rows, columns, and depths (for 3D objects). Can be obtained using `dim()`, plus `nrow()`, `ncol()`, and `ndepth()`.
- `GVectorss`: Vector of two integers indicating number of geometries and number of fields. Can be obtained using `dim()`, plus `nrow()` and `ncol()`.

`extent` Numeric vector with four values (`GSpatial` objects, including `GRegions`, `GRasters`, and `GVectors`): Extent of the object listed in order from westernmost longitude, easternmost longitude, southernmost latitude, northernmost latitude. Can be obtained using `ext()`.

`zextent` Numeric (`GSpatial` objects, including `GRegions`, `GRasters`, and `GVectors`): Bottom- and top-most extents of 3D `GRasters` and `GVectors`. Can be obtained using `zext()`.

`geometry` Character (`GVectorss`): Either `points`, `lines`, or `polygons`. Can be obtained using `geomtype()`.

`nLayers` Integer (`GRasters`): Number of layers ("stacked" rasters–different from number of depths of 3D rasters). Can be obtained using `nlyr()`.

`nGeometries` Integer (`GVectors`): Number of features (points, lines, or polygons). Can be obtained using `nrow()`.

`datatypeGRASS` Character (`GRasters`): Type of data stored in a raster, as interpreted by `GRASS`. This is either `CELL` (integers), `FCELL` (floating-point values), or `DCELL` (double-values). Can be obtained using `datatype()`.

`resolution` Vector of two numeric values (`GRegions`, including `GRasters`): Size of a raster cell in the east-west direction and in the north-south direction. Can be obtained using `res()` and `res3d()`.

`minVal,maxVal` Numeric (`GRasters`): Minimum and maximum value across all cells. Can be obtained using `minmax()`.

`activeCat` Integer (`GRasters`): Column index of the category labels. Must be >0. Note that from the user's standpoint, 1 is subtracted from this number. So a value if `@activeCat` is `2`, then the user would see "1" when printed. Can be obtained using `activeCat()`.

`levels` List of `data.tables` (`GRasters`): Tables for categorical rasters. If a raster is not categorical, the `data.table` is `NULL`, as in `data.table(NULL)`. Can be obtained using `levels()` or `cats()`.

`table` `data.table` (`GVectors`): Table with metadata, one row per geometry (point, line, or plane). If no table is associated with the vector, this must be `data.table(NULL)`. The column with the category value is given in `@catName`.

`catName` Character (`GVectors`): Name of the column in the vector's database that contains category values (integers).

`grassGUI,missing-method`

Start the GRASS GUI (potentially dangerous!)

Description

This function starts the **GRASS** GUI. It is provided merely as a utility... in most cases, it should *not* be used if you are doing any kind of analysis of rasters or vectors using **fasterRaster**. The reason for this prohibition is that **fasterRaster** objects, like `GRasters` and `GVectors`, are really "pointers" to objects in **GRASS**. If **fasterRaster** points to a **GRASS** object that is changed in **GRASS** but not **R**, then **fasterRaster** will not "know" about it, so changed won't be reflected in the **fasterRaster** object.

One aspect of the GUI that is useful but will not change objects is to use it to plot rasters and vectors. However, the a **fasterRaster** object in **R** will have a different name in **GRASS**. The name in **GRASS** of a `GVector` or `GRaster` is given by `sources()`.

Usage

```
## S4 method for signature 'missing'
grassGUI()
```

Value

Nothing (starts the **GRASS** GUI).

See Also

`mow()`

Examples

```
if (grassStarted()) {

# DANGER: Making changes to rasters/vectors in the GUI can "break" them in R.
if (interactive()) grassGUI()

}
```

grassHelp *Open the help page for a GRASS module*

Description

This function opens the manual page for a **GRASS** module (function) in your browser.

Usage

```
grassHelp(x, online = FALSE)
```

Arguments

x	Character: Any of: • The name of a **GRASS** module (e.g., "r.mapcalc"). • "toc": **GRASS** manual table of contents. • "index": Display an index of topics.
online	Logical: If FALSE (default), show the manual page that was included with your installation of **GRASS** on your computer. If FALSE, show the manual page online (requires an Internet connection). In either case, the manual page will display for the version of **GRASS** you have installed.

Value

Nothing (opens a web page).

Examples

```
if (grassStarted() & interactive()) {

# Open help pages for `r.mapcalc` and `r.sun`:
grassHelp("r.mapcalc")
grassHelp("r.sun")

# GRASS table of contents:
grassHelp("toc")

# Index page:
grassHelp("index")

}
```

grassInfo *GRASS citation, version, and copyright information*

Description

Report the **GRASS** citation, version/release year, version number, or copyright information.

Usage

```
grassInfo(x = "citation")
```

Arguments

x	Character: What to return. Any of: • "citation" (default) • "copyright": Copyright information • "version": Version number and release year • "versionNumber": Version number as numeric, major and minor only (e.g., 8.4) Partial matching is used and case is ignored.

Value

Character.

Examples

```
if (grassStarted()) {

# Citation
grassInfo()

# Version number
grassInfo("version")

# Version number
grassInfo("versionNumber")

# Version number
grassInfo("versionNumber")

# Copyright
grassInfo("copyright")

}
```

grassStarted *Has "GRASS" been started or not?*

Description

Returns TRUE or FALSE, depending on whether a **GRASS** connection has been made or not within the current **R** session. Usually used only by developers. **GRASS** is started the first time fast() is used.

Usage

```
grassStarted()
```

Value

Logical.

Examples

```
grassStarted()
```

grid,GRaster-method *Create a grid GVector*

Description

This function creates a GVector of "wall-to-wall" cells (like a lattice). The input can be a GVector or GRaster, which provides the extent of the output.

Usage

```
## S4 method for signature 'GRaster'
grid(x, nx = NULL, ny = NULL, use = "number", angle = 0)

## S4 method for signature 'GVector'
grid(x, nx = NULL, ny = NULL, use = "number", angle = 0)
```

Arguments

x	A GRaster or GVector.
nx, ny	Integer or numeric: • If use is "number", then these values represent the number of rows and columns in the grid.

- If use is size, then these values represent the size of the cells in the x- and y-dimensions.

use — Character: How to generate the grid. If this is number (default), then nx and ny are taken to be the number of grid cells. If size, then nx and ny are taken to be the size of the grid cells.

angle — Numeric: Degrees by which to rotate grid (from north, clockwise).

Value

A GVector.

See Also

hexagons(), module v.mkgrid in **GRASS**

Examples

```
if (grassStarted()) {

# Setup
library(sf)

# Points vector of specimens of species in the plant genus Dypsis
madCoast0 <- fastData("madCoast0")

# Convert sf to a GVector:
coast <- fast(madCoast0)

### grid

# grid specified by number of cells in x-dimension
g1 <- grid(coast, nx = 10)
plot(coast, col = "cornflowerblue")
plot(g1, add = TRUE)

# grid specified by number of cells in x- and y-dimension
g2 <- grid(coast, nx = 10, ny = 5)
plot(coast, col = "cornflowerblue")
plot(g2, add = TRUE)

# grid specified by size of cells in both dimensions
g3 <- grid(coast, nx = 1250, ny = 2000, use = "size")
plot(coast, col = "cornflowerblue")
plot(g3, add = TRUE)

### hexagons

hexes <- hexagons(coast, ny = 10)
plot(hexes)
plot(coast, lwd = 2, add = TRUE)
```

```
hexes <- hexagons(coast, ny = 10, expand = c(0.3, 0.1))
plot(hexes)
plot(coast, lwd = 2, add = TRUE)

}
```

---

head,GVector-method *Return first or last part of the data frame of a GVector*

Description

Return the first or last part of a GVector's data table.

Usage

```
## S4 method for signature 'GVector'
head(x, n = 6L, keepnums = TRUE, ...)

## S4 method for signature 'GVector'
tail(x, n = 6L, keepnums = TRUE, ...)
```

Arguments

x	A GVector.
n	Integer: Number of rows to display.
keepnums	Logical: If no rownames are present, create them. Default is TRUE.
...	Other arguments.

Value

A data.table or data.frame.

See Also

terra::head(), terra::tail()

Examples

```
if (grassStarted()) {

# Setup
library(sf)

# Example data:
madCoast4 <- fastData("madCoast4")
madRivers <- fastData("madRivers")
madDypsis <- fastData("madDypsis")
```

```
# Convert sf vectors to GVectors:
coast <- fast(madCoast4)
rivers <- fast(madRivers)
dypsis <- fast(madDypsis)

# Geographic properties:
ext(rivers) # extent
crs(rivers) # coordinate reference system
st_crs(rivers) # coordinate reference system
coordRef(rivers) # coordinate reference system

# Column names and data types:
names(coast)
datatype(coast)

# Points, lines, or polygons?
geomtype(dypsis)
geomtype(rivers)
geomtype(coast)

is.points(dypsis)
is.points(coast)

is.lines(rivers)
is.lines(dypsis)

is.polygons(coast)
is.polygons(dypsis)

# Number of dimensions:
topology(rivers)
is.2d(rivers) # 2-dimensional?
is.3d(rivers) # 3-dimensional?

# Just the data table:
as.data.frame(rivers)
as.data.table(rivers)

# Top/bottom of the data table:
head(rivers)
tail(rivers)

# Vector or table with just selected columns:
names(rivers)
rivers$NAME
rivers[[c("NAM", "NAME_0")]]
rivers[[c(3, 5)]]

# Select geometries/rows of the vector:
nrow(rivers)
selected <- rivers[2:6]
nrow(selected)
```

```
# Plot:
plot(coast)
plot(rivers, col = "blue", add = TRUE)
plot(selected, col = "red", lwd = 2, add = TRUE)

# Vector math:
hull <- convHull(dypsis)

un <- union(coast, hull)
sameAsUnion <- coast + hull
plot(un)
plot(sameAsUnion)

inter <- intersect(coast, hull)
sameAsIntersect <- coast * hull
plot(inter)
plot(sameAsIntersect)

er <- erase(coast, hull)
sameAsErase <- coast - hull
plot(er)
plot(sameAsErase)

xr <- xor(coast, hull)
sameAsXor <- coast / hull
plot(xr)
plot(sameAsXor)

# Vector area and length:
expanse(coast, unit = "km") # polygons areas
expanse(rivers, unit = "km") # river lengths

### Fill holes

# First, we will make some holes by creating buffers around points.
buffs <- buffer(dypsis, 500)

holes <- coast - buffs
plot(holes)

filled <- fillHoles(holes, fail = FALSE)

}
```

---

hexagons,GRaster-method

Create a hexagonal grid

Description

This function creates a GVector of "wall-to-wall" hexagons. The input can be a GVector or GRaster, which provides the extent of the output.

Usage

```
## S4 method for signature 'GRaster'
hexagons(x, ny = 10, expand = 0, angle = 0)

## S4 method for signature 'GVector'
hexagons(x, ny = 10, expand = 0, angle = 0)
```

Arguments

x	A GRaster or GVector.
ny	Integer or numeric integer: Number of rows of hexagons that span the extent of object x.
expand	One or two numeric values: Expand the region by this proportion in both directions (a single value) or in the x- and y-dimensions separately. Expanding the region can be helpful to ensure the entire area of interest is covered by polygons, which can otherwise leave gaps at the edges. The number of rows and columns will be increased, but the number of hexagons that span x will still be ny.
angle	Numeric: Degrees by which to rotate grid (from north, clockwise).

Value

A GVector.

See Also

grid(), module v.mkgrid in **GRASS**

Examples

```
if (grassStarted()) {

# Setup
library(sf)

# Points vector of specimens of species in the plant genus Dypsis
madCoast0 <- fastData("madCoast0")

# Convert sf to a GVector:
coast <- fast(madCoast0)

### grid

# grid specified by number of cells in x-dimension
g1 <- grid(coast, nx = 10)
```

```
plot(coast, col = "cornflowerblue")
plot(g1, add = TRUE)

# grid specified by number of cells in x- and y-dimension
g2 <- grid(coast, nx = 10, ny = 5)
plot(coast, col = "cornflowerblue")
plot(g2, add = TRUE)

# grid specified by size of cells in both dimensions
g3 <- grid(coast, nx = 1250, ny = 2000, use = "size")
plot(coast, col = "cornflowerblue")
plot(g3, add = TRUE)

### hexagons

hexes <- hexagons(coast, ny = 10)
plot(hexes)
plot(coast, lwd = 2, add = TRUE)

hexes <- hexagons(coast, ny = 10, expand = c(0.3, 0.1))
plot(hexes)
plot(coast, lwd = 2, add = TRUE)

}
```

`hillshade,GRaster-method`
Hillshading

Description

Hillshade rasters are often used for display purposes because they make topographical relief look "real" to the eye.

Usage

```
## S4 method for signature 'GRaster'
hillshade(x, angle = 45, direction = 0, zscale = 1)
```

Arguments

`x`	A `GRaster` (typically representing elevation).
`angle`	Numeric: The altitude of the sun above the horizon in degrees. Valid values are in the range [0, 90], and the default value is 45 (half way from the horizon to overhead).
`direction`	The direction (azimuth) in which the sun is shining in degrees. Valid values are in the range 0 to 360. The default is 0, meaning the sun is at due south (180 degrees) and shining due north (0 degrees). Note that in this function, 0 corresponds to north and 180 to south, but in the **GRASS** module `r.relief`, "east orientation" is used (0 is east, 90 is north, etc.).

zscale Numeric: Value by which to exaggerate terrain. The default is 1. Numbers greater than this will increase apparent relief, and less than this (even negative) will diminish it.

Value

A GRaster.

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster
elev <- fast(madElev)

# Calculate all topographic metrics
topos <- terrain(elev, v = "*")
topos

plot(topos) # NB Aspect has values of NA when it cannot be defined

# Calculate a hillshade raster
hs <- hillshade(elev)
plot(hs)

}
```

---

hist,GRaster-method *Plot a histogram of raster values*

Description

This function creates a histogram of values in GRaster. The function is modeled after graphics::hist(), but actually uses graphics::barplot().

Usage

```
## S4 method for signature 'GRaster'
hist(x, layer, maxnl = 16, bins = 30, freq = TRUE, ...)
```

Arguments

x	A GRaster.
layer	Character, numeric, or integer: Indicates which layer of a multi-layer GRaster for which to plot a histogram. The layer can be identified using its name() (character) or index (numeric or integer). If this is missing, then up to maxnl layers are plotted.
maxnl	Maximum number of layers for which to create histograms. This is 16 by default, but ignored if layer is defined.
bins	Positive numeric integer: Number of bins in which to divide values of a raster with continuous values. For integer and categorical rasters, each value is tallied.
freq	Logical: If TRUE (default), plot the frequency of values. If FALSE, plot the density of values (i.e., the number in each bin divided by the total number of cells with non-NA values).
...	Arguments to pass to graphics::barplot().

Value

A named list of data.frames (invisibly), one per layer plotted, and creates a graph.

Examples

```
if (grassStarted()) {

# Example data
madElev <- fastData("madElev") # elevation raster
madLANDSAT <- fastData("madLANDSAT") # multi-layer raster
madRivers <- fastData("madRivers") # lines vector

# Convert SpatRaster to GRaster and SpatVector to GVector
elev <- fast(madElev)
rivers <- fast(madRivers)
landsat <- fast(madLANDSAT)

# Plot:
plot(elev)
plot(rivers, add = TRUE)

# Histograms:
hist(elev)
hist(landsat)

# Plot surface reflectance in RGB:
plotRGB(landsat, 3, 2, 1) # "natural" color
plotRGB(landsat, 4, 1, 2, stretch = "lin") # emphasize near-infrared (vegetation)

# Make composite map from RGB layers and plot in grayscale:
comp <- compositeRGB(r = landsat[[3]], g = landsat[[2]], b = landsat[[1]])
grays <- paste0("gray", 0:100)
```

```
plot(comp, col = grays)

}
```

`horizonHeight,GRaster-method`
Horizon height

Description

`horizonHeight()` uses a raster representing elevation to calculate the height of the horizon in a particular direction from each cell on a raster. Height is expressed in radians or degrees from the horizontal.

Usage

```
## S4 method for signature 'GRaster'
horizonHeight(
  x,
  units = "radians",
  step = 90,
  northIs0 = TRUE,
  bufferZone = 0,
  distance = 1,
  maxDist = NULL
)
```

Arguments

`x`	A `GRaster`.
`units`	Character: Units of the height. Either `radians` (default) or `degrees`. Partial matching is used.
`step`	Numeric integer between 0 and 360, inclusive: Angle step size (in degrees) for calculating horizon height. The direction in which horizon height is calculated is incremented from 0 to 360, with the last value excluded.
`northIs0`	Logical: If `TRUE` (default), horizon height calculated in the 0-degree direction will be facing north, and proceed clockwise So, under "north orientation", 0 is north, 90 east, 180 south, and 270 west. If `FALSE`, angles are in "east orientation", and proceed counterclockwise from east. So, east is 0, north 90, west 180, and south 270. North orientation is the default for this function in **R**, but east orientation is the default in the **GRASS** module `r.horizon`. **Note:** The `sun()` function requires aspect to be in east orientation.
`bufferZone`	Numeric >= 0 (default is 0): A buffer of the specified width will be generated around the raster before calculation of horizon angle. If the coordinate system is in longitude/latitude (e.g., WGS84 or NAD83), then this is specified in degrees. Otherwise units are map units (usually meters).

distance	Numeric between 0.5 and 1.5, inclusive (default is 1): This determines the step size when searching for the horizon from a given point. The default value of 1 goes cell-by-cell (i.e., search distance step size is one cell width).
maxDist	Either NULL (default) or numeric >= 0: Maximum distance to consider when finding horizon height in meters. If NULL, the maximum distance is the full extent of the raster. Smaller values can decrease run time but also reduce accuracy.

Value

A GRaster with one or more layers. The layers will be named height_*xyz*, where *xyz* is degrees from north or from east, depending on whether north or east orientation is used.

See Also

GRASS manual page for module r.horizon (see grassHelp("r.horizon"))

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")

# convert a SpatRaster to a GRaster
elev <- fast(madElev)

# calculate horizon height in north and east directions
hhNorth <- horizonHeight(elev)
hhNorth
plot(hhNorth)

# calculate horizon height in east and north directions
hhEast <- horizonHeight(elev, northIs0 = FALSE)
hhEast
plot(hhEast)

}
```

---

init,GRaster-method *GRaster with values equal to row, column, coordinate, regular, or "chess"*

Description

This function can be used to make a GRaster with cell values equal to the cell center's longitude, latitude, row, or column, or in a "chess"-like or "regular" pattern.

Usage

```
## S4 method for signature 'GRaster'
init(x, fun, odd = TRUE, vals = c(0, 1))
```

Arguments

x	A GRaster to be used as a template.
fun	Character: Any of: • "x" or "y": Cell longitude or latitude • "row" or "col": Cell row or column • "chess": Alternating values. • "regular": Evenly-spaced cells with the same value.
odd	Logical: If TRUE (default), and fun is "chess", then the top left cell in the raster will be a "negative" cell. If FALSE, then the top left cell with be "positive".
vals	Vector of two numeric values: If fun is "chess" or "regular", then assign the first value to "positive" cells and the second value to "negative" cells. The default is c(1, 0)

Value

A GRaster with as many layers as x.

See Also

terra::init(), longlat()

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Elevation raster, rivers vector
madElev <- fastData("madElev")

# Convert to a GRaster
elev <- fast(madElev)

# Cell coordinates
init(elev, "x")
init(elev, "y")

# Cell row or column
init(elev, "row")
init(elev, "col")

# Chess
elevAgg <- aggregate(elev, 32) # make cells bigger so we can see them
```

```
chessOdd <- init(elevAgg, "chess")
chessEven <- init(elevAgg, "chess", odd = FALSE)

chess <- c(chessOdd, chessEven)
names(chess) <- c("odd", "even")
plot(chess)

# Chess with user-defined values
elevAgg <- aggregate(elev, 32) # make cells bigger so we can see

chessOdd13 <- init(elevAgg, "chess", vals = c(0, 13))
chessEven13 <- init(elevAgg, "chess", odd = FALSE, vals = c(0, 13))

chess13 <- c(chessOdd13, chessEven13)
names(chess13) <- c("odd", "even")
plot(chess13)

# Regular
elevAgg <- aggregate(elev, 32) # make cells bigger so we can see

regOdd <- init(elevAgg, "regular")
regEven <- init(elevAgg, "regular", odd = FALSE)

reg <- c(regOdd, regEven)
names(reg) <- c("odd", "even")
plot(reg)

}
```

`interpIDW,GVector,GRaster-method`

Interpolate values at points to a GRaster using inverse-distance weighting

---

Description

This function interpolates values from a set of points to a raster using inverse distance weighting (IDW).

Usage

```
## S4 method for signature 'GVector,GRaster'
interpIDW(x, y, field, nPoints = Inf, power = 2)
```

Arguments

`x`	A "points" `GVector`.

y	A GRaster to serve as a template for interpolation: Only points in x that fall inside the extent of the raster will be used for interpolation. You can increase the extent of a GRaster using extend().
field	Character, integer, or numeric integer: Name or index of the column in x with values to interpolate. If NULL and if x is a 3-dimensional "points" GVector, then the interpolation will act on the z-coordinate of each point.
nPoints	Integer or numeric integer: Number of nearest points to use for interpolation. The default is to use all points (Inf).
power	Numeric value > 0: Power to which to take distance when interpolating. The default value is two, so the value of each point used for interpolation is $1/d^2$ where d is distance.

Value

A GRaster.

See Also

terra::interpIDW(), interpSplines(), fillNAs(), **GRASS** module v.surf.idw (se grassHelp("v.surf.idw"))

---

interpSplines,GVector,GRaster-method

Interpolate values at points to a GRaster using splines

Description

This function interpolates values in the data table of a "points" GVector to a GRaster using splines with Tykhonov regularization to avoid overfitting.

Usage

```
## S4 method for signature 'GVector,GRaster'
interpSplines(
  x,
  y,
  field,
  method = "bilinear",
  lambda = NULL,
  solver = "Cholesky",
  xlength = NULL,
  ylength = NULL,
  interpolate = TRUE,
  verbose = is.null(lambda)
)
```

Arguments

x	A "points" GVector.
y	A GRaster: The output will have the same extent and resolution as this raster.
field	Character or integer or numeric integer: Name or index of the column in x with values to interpolate. If NULL and if x is a 3-dimensional "points" GVector, then the interpolation will act on the z-coordinate of each point.
method	Character: The method to use for interpolation can be either "bilinear" (default) or "bicubic". Partial matching is used.
lambda	Either NULL (default) or numeric value > 0: The Tykhonov regularization parameter. If NULL, cross-validation will be used to determine the optimal parameter value. Cross-validation can take quite a while. If you use cross-validation, the output will either be a GRaster or a data.frame, depending on the value of interpolate.
solver	Character: Type of solver to use. Can be either of "Cholesky" or "cg". Partial matching is used and case is ignored.
xlength, ylength	Either NULL (default), or numeric > 0: Length of the spline step in the x- and y-directions. If NULL, these will be set to 4 times the length of the extent in the respective direction.
interpolate	Logical: If TRUE (default), then create a GRaster with interpolated values. If FALSE, return a table with lambda values from cross-validation. This argument is ignored if lambda is a numeric value.
verbose	Logical: if TRUE, display progress.

Details

If you receive the error, "No data within this subregion. Consider increasing spline step values, try increasing the values of xlength and ylength.

If cross-validation takes too long, or other warnings/errors persist, you can randomly subsample x to ~100 points to get an optimum value of lambda (using interpolate = FALSE), then use this value in the same function again without cross-validation (setting lambda equal to this value and interpolate = TRUE).

Value

Output depends on values of lambda and interpolate:

- lambda is NULL and interpolate is TRUE: A GRaster with an attribute named lambdas. This is a data.frame with values of lambda that were assessed, plus mean (mean residual value) and rms (root mean square error). You can see the table using attr(output_raster, "lambdas", exact = TRUE).
- lambda is NULL and interpolate is FALSE: A data.frame with values of lambdas that were assessed, plus mean (mean residual value) and rms (root mean square error). You can see the table using attr(output_raster, "lambdas", exact = TRUE).
- lambda is a number (interpolate is ignored): A GRaster.

See Also

interpIDW(), fillNAs(), **GRASS** module v.surf.bspline (see grassHelp("v.surf.bspline"))

---

intersect,GVector,GVector-method
Intersection of two GVectors

Description

The intersect() function selects the area of overlap between two GVectors of the same type (points, lines or polygons). You can also use the * operator (e.g., vect1 * vect2).

Usage

```
## S4 method for signature 'GVector,GVector'
intersect(x, y)
```

Arguments

x, y	GVectors.

Value

A GVector.

See Also

c(), aggregate(), crop(), union(), xor()

Examples

```
if (grassStarted()) {

# Setup
library(sf)

# Polygon of coastal Madagascar and Dypsis specimens
madCoast4 <- fastData("madCoast4") # polygons
madDypsis <- fastData("madDypsis") # points

# Convert vectors:
coast4 <- fast(madCoast4)
dypsis <- fast(madDypsis)

# Create another polygons vector from a convex hull around Dypsis points
hull <- convHull(dypsis)

### union()
```

```
unioned <- union(coast4, hull)
plot(unioned)

plus <- coast4 + hull # same as union()

### intersect

inter <- intersect(coast4, hull)
plot(coast4)
plot(hull, border = "red", add = TRUE)
plot(inter, border = "blue", add = TRUE)

### xor

xr <- xor(coast4, hull)
plot(coast4)
plot(xr, border = "blue", add = TRUE)

### erase

erased <- erase(coast4, hull)
plot(coast4)
plot(erased, border = "blue", add = TRUE)

minus <- coast4 - hull # same as erase()

}
```

`is.2d,GSpatial-method` *Test if a GRaster or GVector is 2- or 3-dimensional*

Description

Test whether a `GRaster` or `GVector` is 2- or 3-dimensional.

Usage

```
## S4 method for signature 'GSpatial'
is.2d(x)

## S4 method for signature 'GSpatial'
is.3d(x)
```

Arguments

`x`	An object that inherits from the `GSpatial` class (i.e., a `GRaster` or `GVector`).

Value

Logical.

See Also

topology()

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Example data
madElev <- fastData("madElev")
madForest2000 <- fastData("madForest2000")
madCoast0 <- fastData("madCoast0")
madRivers <- fastData("madRivers")
madDypsis <- fastData("madDypsis")

### GRaster properties

# convert SpatRasters to GRasters
elev <- fast(madElev)
forest <- fast(madForest2000)

# plot
plot(elev)

dim(elev) # rows, columns, depths, layers
nrow(elev) # rows
ncol(elev) # columns
ndepth(elev) # depths
nlyr(elev) # layers

res(elev) # resolution

ncell(elev) # cells
ncell3d(elev) # cells (3D rasters only)

topology(elev) # number of dimensions
is.2d(elev) # is it 2D?
is.3d(elev) # is it 3D?

minmax(elev) # min/max values

# name of object in GRASS
sources(elev)

# "names" of the object
names(elev)

# coordinate reference system
crs(elev)
```

```
# extent (bounding box)
ext(elev)

# data type
datatype(elev)

# assigning
copy <- elev
copy[] <- pi # assign all cells to the value of pi
copy

# concatenating multiple GRasters
rasts <- c(elev, forest)
rasts

# adding a raster "in place"
add(rasts) <- ln(elev)
rasts

# subsetting
rasts[[1]]
rasts[["madForest2000"]]

# assigning
rasts[[4]] <- elev > 500

# number of layers
nlyr(rasts)

# names
names(rasts)
names(rasts) <- c("elev_meters", "forest", "ln_elev", "high_elevation")
rasts

### GVector properties

# convert sf vectors to GVectors
coast <- fast(madCoast4)
rivers <- fast(madRivers)
dypsis <- fast(madDypsis)

# extent
ext(rivers)

W(rivers) # western extent
E(rivers) # eastern extent
S(rivers) # southern extent
N(rivers) # northern extent
top(rivers) # top extent (NA for 2D rasters like this one)
bottom(rivers) # bottom extent (NA for 2D rasters like this one)

# coordinate reference system
```

```
crs(rivers)
st_crs(rivers)

# column names and data types
names(coast)
datatype(coast)

# name of object in GRASS
sources(rivers)

# points, lines, or polygons?
geomtype(dypsis)
geomtype(rivers)
geomtype(coast)

is.points(dypsis)
is.points(coast)

is.lines(rivers)
is.lines(dypsis)

is.polygons(coast)
is.polygons(dypsis)

# dimensions
nrow(rivers) # how many spatial features
ncol(rivers) # hay many columns in the data frame

# number of geometries and sub-geometries
ngeom(coast)
nsubgeom(coast)

# 2- or 3D
topology(rivers) # dimensionality
is.2d(elev) # is it 2D?
is.3d(elev) # is it 3D?

# Update values from GRASS
# (Reads values from GRASS... will not appear to do anything in this case)
coast <- update(coast)

### operations on GVectors

# convert to data frame
as.data.frame(rivers)
as.data.table(rivers)

# subsetting
rivers[c(1:2, 5)] # select 3 rows/geometries
rivers[-5:-11] # remove rows/geometries 5 through 11
rivers[ , 1] # column 1
rivers[ , "NAM"] # select column
rivers[["NAM"]] # select column
```

```
rivers[1, 2:3] # row/geometry 1 and column 2 and 3
rivers[c(TRUE, FALSE)] # select every other geometry (T/F vector is recycled)
rivers[ , c(TRUE, FALSE)] # select every other column (T/F vector is recycled)

# removing data table
noTable <- dropTable(rivers)
noTable
nrow(rivers)
nrow(noTable)

# Refresh values from GRASS
# (Reads values from GRASS... will not appear to do anything in this case
# since the rivers object is up-to-date):
rivers <- update(rivers)

# Concatenating multiple vectors
rivers2 <- rbind(rivers, rivers)
dim(rivers)
dim(rivers2)

}
```

`is.int,GRaster-method` *Data type of a raster*

Description

In **fasterRaster**, rasters can have three data types: "factor" (categorical rasters), "integer" (integers), "float" (floating point values, accurate to the 6th to 9th decimal places), and "double" (double-precision values, accurate to the 15th to 17th decimal places). The type of raster can be checked with:

- `is.factor()`: The raster will have integer values and categories matched to the integers (see levels()).
- `is.int()`: Are values integers? Note that `is.int()` will return `FALSE` for categorical rasters, even though cell values are technically integers.
- `is.cell()`: Are values integers (`TRUE` for `integer` and categorical rasters).
- `is.float()`: Are values floating-point precision?
- `is.doub()`: Are values double-floating point precision?

Usage

```
## S4 method for signature 'GRaster'
is.int(x)

## S4 method for signature 'GRaster'
is.cell(x)
```

```
## S4 method for signature 'GRaster'
is.float(x)

## S4 method for signature 'GRaster'
is.doub(x)

## S4 method for signature 'GRaster'
is.factor(x)
```

Arguments

x	A GRaster.

Value

Logical.

See Also

datatype(), terra::datatype(), as.int(), as.float(), as.doub(), is.factor(), vignette("GRasters", package = "fasterRaster")

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")
madForest2000 <- fastData("madForest2000")

# Convert SpatRasters to GRasters
elev <- fast(madElev)
forest <- fast(madForest2000)

### GRaster properties

# plotting
plot(elev)

# dimensions
dim(elev) # rows, columns, depths, layers
nrow(elev) # rows
ncol(elev) # columns
ndepth(elev) # depths
nlyr(elev) # layers

res(elev) # resolution (2D)
res3d(elev) # resolution (3D)
zres(elev) # vertical resolution
```

```
xres(elev) # vertical resolution
yres(elev) # vertical resolution
zres(elev) # vertical resolution (NA because this is a 2D GRaster)

# cell counts
ncell(elev) # cells
ncell3d(elev) # cells (3D rasters only)

# number of NA and non-NA cells
nacell(elev)
nonnacell(elev)

# topology
topology(elev) # number of dimensions
is.2d(elev) # is it 2-dimensional?
is.3d(elev) # is it 3-dimensional?

minmax(elev) # min/max values

# "names" of the object
names(elev)

# coordinate reference system
crs(elev)
st_crs(elev)
coordRef(elev)

# extent (bounding box)
ext(elev)

# vertical extent (not defined for this raster)
zext(elev)

# data type
datatype(elev) # fasterRaster type
datatype(elev, "GRASS") # GRASS type
datatype(elev, "terra") # terra type
datatype(elev, "GDAL") # GDAL type

is.integer(elev)
is.float(elev)
is.double(elev)
is.factor(elev)

# convert data type
as.int(elev) # integer; note that "elev" is already of type "integer"
as.float(elev) # floating-precision
as.doub(elev) # double-precision

# assigning
pie <- elev
pie[] <- pi # assign all cells to the value of pi
pie
```

```
# concatenating multiple GRasters
rasts <- c(elev, forest)
rasts

# subsetting
rasts[[1]]
rasts[["madForest2000"]]

# replacing
rasts[[2]] <- 2 * forest
rasts

# adding layers
rasts[[3]] <- elev > 500 # add a layer
rasts <- c(rasts, sqrt(elev)) # add another
add(rasts) <- ln(elev)
rasts

# names
names(rasts)
names(rasts) <- c("elev_meters", "2_x_forest", "high_elevation", "sqrt_elev", "ln_elev")
rasts

# remove a layer
rasts[["2_x_forest"]] <- NULL
rasts

# number of layers
nlyr(rasts)

# correlation and covariance matrices
madLANDSAT <- fastData("madLANDSAT")
landsat <- fast(madLANDSAT) # projects matrix
layerCor(landsat) # correlation
layerCor(landsat, fun = 'cov') # covariance

}
```

---

is.lonlat,character-method

Test if a coordinate reference system is unprojected

Description

`is.lonlat()` attempts to determine if a coordinate reference system is unprojected (e.g., WGS84, NAD83, NAD27, etc.). For `GRasters` and `GVectors`, the function should always be correct. For WKT character strings and `sf` vectors, it does this by looking for the "CONVERSION[" tag in the WKT string (or the object's WKT string), and if it finds one, returns `FALSE`. This may not be truthful in all cases.

Usage

```
## S4 method for signature 'character'
is.lonlat(x)

## S4 method for signature 'GLocation'
is.lonlat(x)

## S4 method for signature 'sf'
is.lonlat(x)
```

Arguments

x	A WKT coordinate reference string or an object from which on can be obtained (e.g., a GRaster, GVector, GRegion, GLocation, SpatRaster, SpatVector, or sf object).

Value

Logical (TRUE if unprojected, FALSE otherwise).

See Also

terra::is.lonlat()

---

is.na,GRaster-method *Mathematical operations on each layer of a GRasters*

Description

You can apply mathematical functions to each layer of a GRaster. The output is a GRaster with the same number or layers as the input. Available functions include:

- NAs:
  - is.na()
  - not.na()
- Absolute value: abs()
- Trigonometric functions (assumes values are in radians):
  - cos()
  - sin()
  - tan()
  - acos()
  - asin()
  - atan()
  - atan2()

- Exponential and logarithmic functions:
  - `exp()`
  - `log()` (natural log)
  - `ln()` (also natural log)
  - `log2()` (log, base 2)
  - `log10()` (log, base 10)
  - `log1p()` (same as `log(x + 1)`)
  - `log10p()` (same as `log(x + 1, base = 10)`)
- Power functions:
  - `sqrt()`
  - `x^y`
- Rounding:
  - `round()`
  - `floor()` (round down)
  - `ceiling()` (round up)
  - `trunc()` (remove decimal portion)

Usage

```
## S4 method for signature 'GRaster'
is.na(x)

## S4 method for signature 'GRaster'
not.na(x, falseNA = FALSE)

## S4 method for signature 'GRaster'
abs(x)

## S4 method for signature 'GRaster'
sin(x)

## S4 method for signature 'GRaster'
cos(x)

## S4 method for signature 'GRaster'
tan(x)

## S4 method for signature 'GRaster'
asin(x)

## S4 method for signature 'GRaster'
acos(x)

## S4 method for signature 'GRaster'
atan(x)
```

```
## S4 method for signature 'GRaster,GRaster'
atan2(y, x)

## S4 method for signature 'GRaster'
exp(x)

## S4 method for signature 'GRaster'
log1p(x)

## S4 method for signature 'GRaster'
log10p(x)

## S4 method for signature 'GRaster'
log(x, base = exp(1))

## S4 method for signature 'GRaster'
ln(x)

## S4 method for signature 'GRaster'
log2(x)

## S4 method for signature 'GRaster'
log10(x)

## S4 method for signature 'GRaster'
sqrt(x)

## S4 method for signature 'GRaster'
round(x, digits = 0)

## S4 method for signature 'GRaster'
floor(x)

## S4 method for signature 'GRaster'
ceiling(x)

## S4 method for signature 'GRaster'
trunc(x)
```

Arguments

x, y	GRasters.
falseNA	Logical (function not.na()): If FALSE (default), non-NA cells will be converted to 1, and NA cells to 0. If TRUE, non-NA cells will be converted to and NA cells will stay as NA.
base	Numeric: Base of the logarithm.
digits	Numeric: Number of digits to round to. If negative, then rounding is to the nearest positive power of 10. For example, if digits = -2, then the GRaster

values are rounded to the nearest 100.

Value

A GRaster.

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Example data
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster
elev <- fast(madElev)
elevs <- c(elev, elev, log10(elev) - 1, sqrt(elev))
names(elevs) <- c("elev1", "elev2", "log_elev", "sqrt_elev")

elev
elevs

# do some math
elev + 100
elev - 100
elev * 100
elev / 100
elev ^ 2
elev %/% 100 # divide then round down
elev %% 100 # modulus

100 + elev
100 %/% elev
100 %% elev

elevs + 100
100 + elevs

# math with logicals
elev + TRUE
elev - TRUE
elev * TRUE
elev / TRUE
elev ^ TRUE
elev %/% TRUE # divide then round down
elev %% TRUE # modulus

elevs + TRUE
TRUE + elevs
```

```
# Raster interacting with raster(s):
elev + elev
elev - elev
elev * elev
elev / elev
elev ^ log(elev)
elev %/% sqrt(elev) # divide then round down
elev %% sqrt(elev) # modulus

elevs + elev
elev * elevs

# sign
abs(-1 * elev)
abs(elevs)

# powers
sqrt(elevs)

# trigonometry
sin(elev)
cos(elev)
tan(elev)

asin(elev)
acos(elev)
atan(elev)

atan(elevs)
atan2(elev, elev^1.2)
atan2(elevs, elev^1.2)
atan2(elev, elevs^1.2)
atan2(elevs, elevs^1.2)

# logarithms
exp(elev)
log(elev)
ln(elev)
log2(elev)
log1p(elev)
log10(elev)
log10p(elev)
log(elev, 3)

log(elevs)

# rounding
round(elev + 0.5)
floor(elev + 0.5)
ceiling(elev + 0.5)
trunc(elev + 0.5)

}
```

kernel,GVector-method *Kernel density estimator of points*

Description

kernel() creates a raster using a kernel density estimator of the density of points in a "points" GVector.

Usage

```
## S4 method for signature 'GVector'
kernel(x, y, kernel = "Epanechnikov", optimize = TRUE, h = NULL)
```

Arguments

x	A "points" GVector.
y	A GRaster: The extent and resolution of this raster will be used to create the density raster. Otherwise, values in this raster are ignored.
kernel	Character: Name of the kernel function to use. Possible values include: • "Epanechnikov" (default) • "Gaussian" • "uniform" • "triangular" • "quartic" • "triweight" • "cosine" Partial matching is used, and case is ignored.
optimize	Logical: If TRUE (default), then attempt to find the optimal radius less than or equal to the radius value using the "Gaussian" kernel. If FALSE, use the radius value as-is.
h	Numeric or NULL (default): Smoothing bandwidth of kernel estimator. If this is NULL, the Epanechnikov kernel is used, and optimize is TRUE, then Silverman's rule-of-thumb is used to estimate the optimal value of h: $$h = 0.9 * min(\sigma_x/n^1/6, \sigma_y/n^1/6)$$ If the Gaussian kernel is used, and optimize is TRUE, then the **GRASS** v.kernel function will attempt to identify the optimal bandwidth, up to the value of h, if h is defined. Otherwise, if h is NULL, then the value will be arbitrarily set at 1/5th of the shorter of the distance of the x- and y-extent of the points.

Value

A GRaster.

See Also

GRASS manual page for module `v.kernel` (see `grassHelp("v.kernel")`)

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Elevation raster, plant specimen collections, rivers vector,
# outline of area vector
madElev <- fastData("madElev")
madDypsis <- fastData("madDypsis")

# Convert to fasterRaster format:
elev <- fast(madElev)
dypsis <- fast(madDypsis)

# Kernel density estimation:
kde <- kernel(dypsis, elev)
plot(kde)
plot(dypsis, add = TRUE, pch = 1)

}
```

---

layerCor,GRaster-method

Correlation between GRasters

Description

This function returns a correlation or covariance matrix between two or more `GRaster` layers. This function returns the sample correlation and covariance (i.e., the denominator is n - 1).

Usage

```
## S4 method for signature 'GRaster'
layerCor(x, fun = "cor")
```

Arguments

`x`	A `GRaster` with two or more layers.
`fun`	Character: Name of the statistic to calculate; either `"cor"` (default) or `"cov"`.

Value

A numeric `matrix`.

See Also

terra::layerCor(), stats::cor(), stats::cov()

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madChelsa <- fastData("madChelsa")

# Convert a SpatRaster to a GRaster:
chelsa <- fast(madChelsa)

# Correlation
layerCor(chelsa, "cor")

# Covariance
layerCor(chelsa, "cov")

}
```

---

levels,GRaster-method *Set and get categories for categorical rasters*

Description

GRasters can represent categorical data. Cell values are actually integers, each corresponding to a category, such as "desert" or "wetland." A categorical raster is associated with a table that matches each value to a category name. The table must be NULL (i.e., no categories–so not a categorical raster), or have at least two columns. The first column must have integers and represent raster values. One or more subsequent columns must have category labels. The column with these labels is the "active category".

- levels(): Displays the "levels" table of a raster (just the value and active category columns).
- cats(): Displays the entire "levels" table of a raster.
- levels()<-: (Re)assigns the "levels" table to each layer of a raster. Assigning a "levels" table to an integer raster makes it a categorical raster.
- categories(): (Re)assigns the "levels" table to specific layer(s) of a raster.
- For a complete list of functions relevant to categorical rasters, see ‘vignette("GRasters", package = "fasterRaster")).

Usage

```
## S4 method for signature 'GRaster'
levels(x)

## S4 method for signature 'GRaster'
cats(x, layer = 1:nlyr(x))

## S4 method for signature 'GRaster'
categories(x, layer = 1, value, active = 1)

## S4 replacement method for signature 'GRaster,data.frame'
levels(x) <- value

## S4 replacement method for signature 'GRaster,data.table'
levels(x) <- value

## S4 replacement method for signature 'GRaster,GRaster'
levels(x) <- value

## S4 replacement method for signature 'GRaster,list'
levels(x) <- value
```

Arguments

x	A GRaster.
layer	Numeric integers, logical vector, or character: For cats() and categories(), this specifies the layer(s)for which to obtain level(s).
value	A data.frame, data.table, a list of data.frames or data.tables with one per raster layer, or a categorical SpatRaster. The table's first column is the "value" column and must contain numeric values (of class numeric or character). If a SpatRaster is supplied, then its categories will be transferred to the GRaster.
active	An integer or a character: The index or column name of the column used for category labels (the "active column"). Following terra::activeCat(), the first column of the "levels" table is ignored, so a value of 1 means to use the second column of the table for labels. A value of 2 means to use the third column, and so on.

Value

Values returned are:

- levels() and cats(): A list of data.frames or data.tables, one per raster layer.
- levels()<- and categories(): A GRaster.

See Also

terra::levels(), levels<-, terra::cats(), terra::categories(), see vignette("GRasters", package = "fasterRaster")

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data: Land cover raster
madCover <- fastData("madCover")

# Convert categorical SpatRaster to categorical GRaster:
cover <- fast(madCover)

### Properties of categorical rasters

cover # note categories
is.factor(cover) # Is the raster categorical?
nlevels(cover) # number of levels
levels(cover) # just the value and active column
cats(cover) # all columns
minmax(cover) # min/max values
minmax(cover, levels = TRUE) # min/max categories
catNames(cover) # column names of the levels table
missingCats(cover) # categories in table with no values in raster
freq(cover) # frequency of each category (number of cells)
zonalGeog(cover) # geometric statistics

### Active column

# Which column sets the category labels?
activeCat(cover)
activeCat(cover, names = TRUE)

activeCats(c(cover, cover))

# Choose a different column for category labels:
levels(cover)
activeCat(cover) <- 2
levels(cover)

### Managing levels tables

# Remove unused levels:
nlevels(cover)
cover <- droplevels(cover)
nlevels(cover)

# Re-assign levels:
value <- c(20, 30, 40, 50, 120, 130, 140, 170)
label <- c("Cropland", "Cropland", "Forest", "Forest",
 "Grassland", "Shrubland", "Herbaceous", "Flooded")

newCats <- data.frame(value = value, label = label)
```

```
cover <- categories(cover, layer = 1, value = newCats)
cats(cover)

# This is the same as:
levels(cover) <- newCats
cats(cover)

# Are there any values not assigned a category?
missingCats(cover)

# Let's assign a category for value 210 (water):
water <- data.frame(value = 210, label = "Water")
addCats(cover) <- water
levels(cover)

# Add more information to the levels table using merge():
landType <- data.frame(
     Value = c(20, 30, 40, 50, 120),
     Type = c("Irrigated", "Rainfed", "Broadleaf evergreen",
     "Broadleaf deciduous", "Mosaic with forest")
)
cats(cover)
cover <- addCats(cover, landType, merge = TRUE)
cats(cover)

### Logical operations on categorical rasters

cover < "Forest" # 1 for cells with a value < 40, 0 otherwise
cover <= "Forest" # 1 for cells with a value < 120, 0 otherwise
cover == "Forest" # 1 for cells with value of 40-120, 0 otherwise
cover != "Forest" # 1 for cells with value that is not 40-120, 0 otherwise
cover > "Forest" # 1 for cells with a value > 120, 0 otherwise
cover >= "Forest" # 1 for cells with a value >= 120, 0 otherwise

cover %in% c("Cropland", "Forest") # 1 for cropland/forest cells, 0 otherwise

### Combine categories from different rasters

# For the example, will create a second categorical raster fromm elevation.

# Divide elevation raster into "low/medium/high" levels:
madElev <- fastData("madElev")
elev <- fast(madElev)
elev <- project(elev, cover, method = "near") # convert to same CRS
fun <- "= if(madElev < 100, 0, if(madElev < 400, 1, 2))"
elevCat <- app(elev, fun)

levs <- data.frame(
     value = c(0, 1, 2),
     elevation = c("low", "medium", "high")
)
levels(elevCat) <- list(levs)
```

```
# Combine levels:
combined <- concats(cover, elevCat)
combined
levels(combined)

# Combine levels, treating value/NA combinations as new categories:
combinedNA <- concats(cover, elevCat, na.rm = FALSE)
combinedNA
levels(combinedNA)

}
```

Logic,GRaster,GRaster-method

Logic-methods operations on GRasters

Description

You can do logical operations on GRasters. A cell with a value of 1 is interpreted as TRUE, and a value of 0 is interpreted as FALSE. You can compare:

- A GRaster to another GRaster
- A GRaster to a logical value (TRUE or FALSE, but not NA–see not.na())
- A GRaster to a numeric or integer value that is 0 or 1

Operators include:

- |: TRUE if either condition is TRUE (or 1), but returns NA if either condition is NA.
- &: TRUE if both conditions are TRUE (or 1), but NA if either is NA.

Usage

```
## S4 method for signature 'GRaster,GRaster'
Logic(e1, e2)

## S4 method for signature 'logical,GRaster'
Logic(e1, e2)

## S4 method for signature 'GRaster,logical'
Logic(e1, e2)

## S4 method for signature 'GRaster,numeric'
Logic(e1, e2)

## S4 method for signature 'numeric,GRaster'
Logic(e1, e2)
```

```
## S4 method for signature 'GRaster,integer'
Logic(e1, e2)

## S4 method for signature 'integer,GRaster'
Logic(e1, e2)
```

Arguments

e1, e2	Two GRasters, or a GRaster and a logical value (TRUE or FALSE, but not NA), a numeric value that is 0 or 1 (but not NA_real_), or an integer value that is 0 or 1 (but not NA_integer_).

Value

A binary GRaster (1 ==> TRUE, 0 ==> FALSE, plus NA when comparison results in NA).

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster
elev <- fast(madElev)
elevs <- c(elev, elev, log10(elev) - 1, sqrt(elev))
names(elevs) <- c("elev1", "elev2", "log_elev", "sqrt_elev")

elev
elevs

# Comparisons
elev < 100
elev <= 100
elev == 100
elev != 100
elev > 100
elev >= 100

elev + 100 < 2 * elev

elevs > 10
10 > elevs

# logic
elev < 10 | elev > 200
elev < 10 | cos(elev) > 0.9

elev < 10 | TRUE
```

```
TRUE | elev > 200

elev < 10 | FALSE
FALSE | elev > 200

elev < 10 & cos(elev) > 0.9

elev < 10 & TRUE
TRUE & elev > 200

elev < 10 & FALSE
FALSE & elev > 200

}
```

`longlat,GRaster-method`

Create longitude/latitude rasters

Description

`longlat()` creates two rasters, one with cell values equal to the longitude of the cell centers, and one with cell values equal to the latitude of the cell centers.

Usage

```
## S4 method for signature 'GRaster'
longlat(x, degrees = TRUE)
```

Arguments

`x`	A `GRaster`.
`degrees`	Logical: If `TRUE` (default), coordinate values of cells will be in degrees. If `FALSE`, and `x` is in a projected coordinate reference system, values will represent coordinates in map units (usually meters). Values will always be in degrees when the coordinate reference system is unprojected (e.g., WGS84, NAD83, etc.).

Value

A `GRaster` stack.

See Also

`init()`

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster
elev <- fast(madElev)

# Create longitude/latitude rasters
ll <- longlat(elev)
ll # note units of cell values!

}
```

`madChelsa` *Rasters of bioclimatic variables for an eastern portion of Madagascar*

Description

Rasters of bioclimatic variables for an eastern portion of Madagascar from CHELSA version 2.1 in unprojected (WGS84) coordinates. Values represent averages across 1980-2010. Only these BIOCLIM variables are included: * `bio1`: Mean annual temperature (deg C) * `bio7`: Temperature annual range (hottest - coldest month temperature; deg C) * `bio12`: Total annual precipitation (mm) * `bio15`: Precipitation seasonality (unit-less)

Format

An object of class `SpatRaster` in unprojected (WGS84) coordinates.

Source

doi:10.1038/sdata.2017.122

References

Karger, D.N., Conrad, O., Bohner, J., Kawohl, T., Kreft, H., Soria-Auza, R.W., Zimmermann, N.E., Linder, H.P., and Kessler, M. 2017. Climatologies at high resolution for the earth's land surface areas. *Scientific Data* 4:170122. doi:10.1038/sdata.2017.122

Examples

```
### vector data

library(sf)
```

```
# For vector data, we can use data(*) or fastData(*):
data(madCoast0) # same as next line
madCoast0 <- fastData("madCoast0") # same as previous
madCoast0
plot(st_geometry(madCoast0))

madCoast4 <- fastData("madCoast4")
madCoast4
plot(st_geometry(madCoast4), add = TRUE)

madRivers <- fastData("madRivers")
madRivers
plot(st_geometry(madRivers), col = "blue", add = TRUE)

madDypsis <- fastData("madDypsis")
madDypsis
plot(st_geometry(madDypsis), col = "red", add = TRUE)

### raster data

library(terra)

# For raster data, we can get the file directly or using fastData(*):
rastFile <- system.file("extdata/madElev.tif", package="fasterRaster")
madElev <- terra::rast(rastFile)

madElev <- fastData("madElev") # same as previous two lines
madElev
plot(madElev)

madForest2000 <- fastData("madForest2000")
madForest2000
plot(madForest2000)

madForest2014 <- fastData("madForest2014")
madForest2014
plot(madForest2014)

# multi-layer rasters
madChelsa <- fastData("madChelsa")
madChelsa
plot(madChelsa)

madPpt <- fastData("madPpt")
madTmin <- fastData("madTmin")
madTmax <- fastData("madTmax")
madPpt
madTmin
madTmax


# RGB raster
madLANDSAT <- fastData("madLANDSAT")
```

```
madLANDSAT
plotRGB(madLANDSAT, 4, 1, 2, stretch = "lin")

# categorical raster
madCover <- fastData("madCover")
madCover
madCover <- droplevels(madCover)
levels(madCover) # levels in the raster
nlevels(madCover) # number of categories
catNames(madCover) # names of categories table

plot(madCover)
```

`madCoast` *Shapefile of a portion of the coastline of Madagascar*

Description

Borders of a selected portion of Madagascar

Format

ESRI Shapefile.

Source

Database of Global Administrative Areas Version 2.8 (GADM)

Examples

```
### vector data

library(sf)

# For vector data, we can use data(*) or fastData(*):
data(madCoast0) # same as next line
madCoast0 <- fastData("madCoast0") # same as previous
madCoast0
plot(st_geometry(madCoast0))

madCoast4 <- fastData("madCoast4")
madCoast4
plot(st_geometry(madCoast4), add = TRUE)

madRivers <- fastData("madRivers")
madRivers
plot(st_geometry(madRivers), col = "blue", add = TRUE)

madDypsis <- fastData("madDypsis")
madDypsis
```

```
plot(st_geometry(madDypsis), col = "red", add = TRUE)

### raster data

library(terra)

# For raster data, we can get the file directly or using fastData(*):
rastFile <- system.file("extdata/madElev.tif", package="fasterRaster")
madElev <- terra::rast(rastFile)

madElev <- fastData("madElev") # same as previous two lines
madElev
plot(madElev)

madForest2000 <- fastData("madForest2000")
madForest2000
plot(madForest2000)

madForest2014 <- fastData("madForest2014")
madForest2014
plot(madForest2014)

# multi-layer rasters
madChelsa <- fastData("madChelsa")
madChelsa
plot(madChelsa)

madPpt <- fastData("madPpt")
madTmin <- fastData("madTmin")
madTmax <- fastData("madTmax")
madPpt
madTmin
madTmax


# RGB raster
madLANDSAT <- fastData("madLANDSAT")
madLANDSAT
plotRGB(madLANDSAT, 4, 1, 2, stretch = "lin")

# categorical raster
madCover <- fastData("madCover")
madCover
madCover <- droplevels(madCover)
levels(madCover) # levels in the raster
nlevels(madCover) # number of categories
catNames(madCover) # names of categories table

plot(madCover)
```

---

madCoast0 *Spatial vector of a portion of the coastline of Madagascar*

Description

Borders of a selected portion of Madagascar

Usage

```
data(madCoast4)
```

Format

An object of class `sf`.

Source

Database of Global Administrative Areas Version 2.8 (GADM)

Examples

```
### vector data

library(sf)

# For vector data, we can use data(*) or fastData(*):
data(madCoast0) # same as next line
madCoast0 <- fastData("madCoast0") # same as previous
madCoast0
plot(st_geometry(madCoast0))

madCoast4 <- fastData("madCoast4")
madCoast4
plot(st_geometry(madCoast4), add = TRUE)

madRivers <- fastData("madRivers")
madRivers
plot(st_geometry(madRivers), col = "blue", add = TRUE)

madDypsis <- fastData("madDypsis")
madDypsis
plot(st_geometry(madDypsis), col = "red", add = TRUE)

### raster data

library(terra)

# For raster data, we can get the file directly or using fastData(*):
rastFile <- system.file("extdata/madElev.tif", package="fasterRaster")
madElev <- terra::rast(rastFile)

madElev <- fastData("madElev") # same as previous two lines
madElev
plot(madElev)

madForest2000 <- fastData("madForest2000")
```

```
madForest2000
plot(madForest2000)

madForest2014 <- fastData("madForest2014")
madForest2014
plot(madForest2014)

# multi-layer rasters
madChelsa <- fastData("madChelsa")
madChelsa
plot(madChelsa)

madPpt <- fastData("madPpt")
madTmin <- fastData("madTmin")
madTmax <- fastData("madTmax")
madPpt
madTmin
madTmax


# RGB raster
madLANDSAT <- fastData("madLANDSAT")
madLANDSAT
plotRGB(madLANDSAT, 4, 1, 2, stretch = "lin")

# categorical raster
madCover <- fastData("madCover")
madCover
madCover <- droplevels(madCover)
levels(madCover) # levels in the raster
nlevels(madCover) # number of categories
catNames(madCover) # names of categories table

plot(madCover)
```

`madCoast4` *Spatial vector of a portion of the coastline of Madagascar*

Description

Borders of a selected portion of Madagascar

Usage

```
data(madCoast4)
```

Format

An object of class `sf`.

Source

Database of Global Administrative Areas Version 2.8 (GADM)

Examples

```
### vector data

library(sf)

# For vector data, we can use data(*) or fastData(*):
data(madCoast0) # same as next line
madCoast0 <- fastData("madCoast0") # same as previous
madCoast0
plot(st_geometry(madCoast0))

madCoast4 <- fastData("madCoast4")
madCoast4
plot(st_geometry(madCoast4), add = TRUE)

madRivers <- fastData("madRivers")
madRivers
plot(st_geometry(madRivers), col = "blue", add = TRUE)

madDypsis <- fastData("madDypsis")
madDypsis
plot(st_geometry(madDypsis), col = "red", add = TRUE)

### raster data

library(terra)

# For raster data, we can get the file directly or using fastData(*):
rastFile <- system.file("extdata/madElev.tif", package="fasterRaster")
madElev <- terra::rast(rastFile)

madElev <- fastData("madElev") # same as previous two lines
madElev
plot(madElev)

madForest2000 <- fastData("madForest2000")
madForest2000
plot(madForest2000)

madForest2014 <- fastData("madForest2014")
madForest2014
plot(madForest2014)

# multi-layer rasters
madChelsa <- fastData("madChelsa")
madChelsa
plot(madChelsa)
```

```
madPpt <- fastData("madPpt")
madTmin <- fastData("madTmin")
madTmax <- fastData("madTmax")
madPpt
madTmin
madTmax


# RGB raster
madLANDSAT <- fastData("madLANDSAT")
madLANDSAT
plotRGB(madLANDSAT, 4, 1, 2, stretch = "lin")

# categorical raster
madCover <- fastData("madCover")
madCover
madCover <- droplevels(madCover)
levels(madCover) # levels in the raster
nlevels(madCover) # number of categories
catNames(madCover) # names of categories table

plot(madCover)
```

`madCover` *Raster of land cover for an eastern portion of Madagascar*

Description

Raster of land cover for an eastern portion of Madagascar. Note that the land cover classes have been simplified, so this raster should *not* be used for "real" analyses.

Format

An object of class `SpatRaster` in unprojected (WGS84) coordinates.

Source

http://due.esrin.esa.int

References

Arino O., P. Bicheron, F. Achard, J. Latham, R. Witt and J.-L. Weber. 2008. GlobCover: The most detailed portrait of Earth. European Space Agency Bulletin 136:25-31. http://due.esrin.esa.int.

See Also

madCoverCats, `vignette("GRasters", package = "fasterRaster")`

Examples

```
### vector data

library(sf)

# For vector data, we can use data(*) or fastData(*):
data(madCoast0) # same as next line
madCoast0 <- fastData("madCoast0") # same as previous
madCoast0
plot(st_geometry(madCoast0))

madCoast4 <- fastData("madCoast4")
madCoast4
plot(st_geometry(madCoast4), add = TRUE)

madRivers <- fastData("madRivers")
madRivers
plot(st_geometry(madRivers), col = "blue", add = TRUE)

madDypsis <- fastData("madDypsis")
madDypsis
plot(st_geometry(madDypsis), col = "red", add = TRUE)

### raster data

library(terra)

# For raster data, we can get the file directly or using fastData(*):
rastFile <- system.file("extdata/madElev.tif", package="fasterRaster")
madElev <- terra::rast(rastFile)

madElev <- fastData("madElev") # same as previous two lines
madElev
plot(madElev)

madForest2000 <- fastData("madForest2000")
madForest2000
plot(madForest2000)

madForest2014 <- fastData("madForest2014")
madForest2014
plot(madForest2014)

# multi-layer rasters
madChelsa <- fastData("madChelsa")
madChelsa
plot(madChelsa)

madPpt <- fastData("madPpt")
madTmin <- fastData("madTmin")
madTmax <- fastData("madTmax")
madPpt
```

```
madTmin
madTmax


# RGB raster
madLANDSAT <- fastData("madLANDSAT")
madLANDSAT
plotRGB(madLANDSAT, 4, 1, 2, stretch = "lin")

# categorical raster
madCover <- fastData("madCover")
madCover
madCover <- droplevels(madCover)
levels(madCover) # levels in the raster
nlevels(madCover) # number of categories
catNames(madCover) # names of categories table

plot(madCover)
```

`madCoverCats` *Table of land cover classes for an eastern portion of Madagascar*

Description

This data frame corresponds to the madCover raster, which represents land cover for an eastern portion of Madagascar. Note that the land cover classes have been simplified, so this table and raster should *not* be used for "real" analyses.

Format

An object of class `data.frame`.

Source

http://due.esrin.esa.int

References

Arino O., P. Bicheron, F. Achard, J. Latham, R. Witt and J.-L. Weber. 2008. GlobCover: The most detailed portrait of Earth. European Space Agency Bulletin 136:25-31. http://due.esrin.esa.int.

See Also

madCover, `vignette("GRasters", package = "fasterRaster")`

Examples

```
### vector data

library(sf)

# For vector data, we can use data(*) or fastData(*):
data(madCoast0) # same as next line
madCoast0 <- fastData("madCoast0") # same as previous
madCoast0
plot(st_geometry(madCoast0))

madCoast4 <- fastData("madCoast4")
madCoast4
plot(st_geometry(madCoast4), add = TRUE)

madRivers <- fastData("madRivers")
madRivers
plot(st_geometry(madRivers), col = "blue", add = TRUE)

madDypsis <- fastData("madDypsis")
madDypsis
plot(st_geometry(madDypsis), col = "red", add = TRUE)

### raster data

library(terra)

# For raster data, we can get the file directly or using fastData(*):
rastFile <- system.file("extdata/madElev.tif", package="fasterRaster")
madElev <- terra::rast(rastFile)

madElev <- fastData("madElev") # same as previous two lines
madElev
plot(madElev)

madForest2000 <- fastData("madForest2000")
madForest2000
plot(madForest2000)

madForest2014 <- fastData("madForest2014")
madForest2014
plot(madForest2014)

# multi-layer rasters
madChelsa <- fastData("madChelsa")
madChelsa
plot(madChelsa)

madPpt <- fastData("madPpt")
madTmin <- fastData("madTmin")
madTmax <- fastData("madTmax")
madPpt
```

```
madTmin
madTmax


# RGB raster
madLANDSAT <- fastData("madLANDSAT")
madLANDSAT
plotRGB(madLANDSAT, 4, 1, 2, stretch = "lin")

# categorical raster
madCover <- fastData("madCover")
madCover
madCover <- droplevels(madCover)
levels(madCover) # levels in the raster
nlevels(madCover) # number of categories
catNames(madCover) # names of categories table

plot(madCover)
```

`madDypsis` *Spatial points vector of records of Dypsis in eastern Madagascar*

Description

Spatial points vector of herbarium specimens and observations of plants in the genus *Dypsis* (slender, evergreen palms) from a portion of eastern Madagascar.

Usage

```
data(madDypsis)
```

Format

An object of class `sf`.

Source

Global Biodiversity Information Facility (GBIF)

Examples

```
### vector data

library(sf)

# For vector data, we can use data(*) or fastData(*):
data(madCoast0) # same as next line
madCoast0 <- fastData("madCoast0") # same as previous
madCoast0
plot(st_geometry(madCoast0))
```

```
madCoast4 <- fastData("madCoast4")
madCoast4
plot(st_geometry(madCoast4), add = TRUE)

madRivers <- fastData("madRivers")
madRivers
plot(st_geometry(madRivers), col = "blue", add = TRUE)

madDypsis <- fastData("madDypsis")
madDypsis
plot(st_geometry(madDypsis), col = "red", add = TRUE)

### raster data

library(terra)

# For raster data, we can get the file directly or using fastData(*):
rastFile <- system.file("extdata/madElev.tif", package="fasterRaster")
madElev <- terra::rast(rastFile)

madElev <- fastData("madElev") # same as previous two lines
madElev
plot(madElev)

madForest2000 <- fastData("madForest2000")
madForest2000
plot(madForest2000)

madForest2014 <- fastData("madForest2014")
madForest2014
plot(madForest2014)

# multi-layer rasters
madChelsa <- fastData("madChelsa")
madChelsa
plot(madChelsa)

madPpt <- fastData("madPpt")
madTmin <- fastData("madTmin")
madTmax <- fastData("madTmax")
madPpt
madTmin
madTmax


# RGB raster
madLANDSAT <- fastData("madLANDSAT")
madLANDSAT
plotRGB(madLANDSAT, 4, 1, 2, stretch = "lin")

# categorical raster
madCover <- fastData("madCover")
```

```
madCover
madCover <- droplevels(madCover)
levels(madCover) # levels in the raster
nlevels(madCover) # number of categories
catNames(madCover) # names of categories table

plot(madCover)
```

---

madElev *Elevation raster for an eastern portion of Madagascar*

Description

Elevation raster for an eastern portion of Madagascar.

Format

An object of class `SpatRaster`. Values are mean meters above sea level.

Source

WorldClim Version 2.1

Examples

```
### vector data

library(sf)

# For vector data, we can use data(*) or fastData(*):
data(madCoast0) # same as next line
madCoast0 <- fastData("madCoast0") # same as previous
madCoast0
plot(st_geometry(madCoast0))

madCoast4 <- fastData("madCoast4")
madCoast4
plot(st_geometry(madCoast4), add = TRUE)

madRivers <- fastData("madRivers")
madRivers
plot(st_geometry(madRivers), col = "blue", add = TRUE)

madDypsis <- fastData("madDypsis")
madDypsis
plot(st_geometry(madDypsis), col = "red", add = TRUE)

### raster data

library(terra)
```

```
# For raster data, we can get the file directly or using fastData(*):
rastFile <- system.file("extdata/madElev.tif", package="fasterRaster")
madElev <- terra::rast(rastFile)

madElev <- fastData("madElev") # same as previous two lines
madElev
plot(madElev)

madForest2000 <- fastData("madForest2000")
madForest2000
plot(madForest2000)

madForest2014 <- fastData("madForest2014")
madForest2014
plot(madForest2014)

# multi-layer rasters
madChelsa <- fastData("madChelsa")
madChelsa
plot(madChelsa)

madPpt <- fastData("madPpt")
madTmin <- fastData("madTmin")
madTmax <- fastData("madTmax")
madPpt
madTmin
madTmax


# RGB raster
madLANDSAT <- fastData("madLANDSAT")
madLANDSAT
plotRGB(madLANDSAT, 4, 1, 2, stretch = "lin")

# categorical raster
madCover <- fastData("madCover")
madCover
madCover <- droplevels(madCover)
levels(madCover) # levels in the raster
nlevels(madCover) # number of categories
catNames(madCover) # names of categories table

plot(madCover)
```

`madForest2000` *Forest cover in year 2000 for a portion of Madagascar*

Description

Raster of occurrence/non-occurrence of forest cover in a portion of Madagascar. Cells are 30-m in resolution. Values represent forest (1) or non-forest (NA).

Format

An object of class SpatRaster.

References

Vielledent, G., Grinand, C., Rakotomala, F.A., Ranaivosoa, **R**., Rakotoarijaona, J-R., Allnutt, T.F., and Achard, F. 2018. Combining global tree cover loss data with historical national forest cover maps to look at six decades of deforestation and forest fragmentation in Madagascar. *Biological Conservation* 222:189-197. doi:10.1016/j.biocon.2018.04.008.

Examples

```
### vector data

library(sf)

# For vector data, we can use data(*) or fastData(*):
data(madCoast0) # same as next line
madCoast0 <- fastData("madCoast0") # same as previous
madCoast0
plot(st_geometry(madCoast0))

madCoast4 <- fastData("madCoast4")
madCoast4
plot(st_geometry(madCoast4), add = TRUE)

madRivers <- fastData("madRivers")
madRivers
plot(st_geometry(madRivers), col = "blue", add = TRUE)

madDypsis <- fastData("madDypsis")
madDypsis
plot(st_geometry(madDypsis), col = "red", add = TRUE)

### raster data

library(terra)

# For raster data, we can get the file directly or using fastData(*):
rastFile <- system.file("extdata/madElev.tif", package="fasterRaster")
madElev <- terra::rast(rastFile)

madElev <- fastData("madElev") # same as previous two lines
madElev
plot(madElev)
```

```
madForest2000 <- fastData("madForest2000")
madForest2000
plot(madForest2000)

madForest2014 <- fastData("madForest2014")
madForest2014
plot(madForest2014)

# multi-layer rasters
madChelsa <- fastData("madChelsa")
madChelsa
plot(madChelsa)

madPpt <- fastData("madPpt")
madTmin <- fastData("madTmin")
madTmax <- fastData("madTmax")
madPpt
madTmin
madTmax


# RGB raster
madLANDSAT <- fastData("madLANDSAT")
madLANDSAT
plotRGB(madLANDSAT, 4, 1, 2, stretch = "lin")

# categorical raster
madCover <- fastData("madCover")
madCover
madCover <- droplevels(madCover)
levels(madCover) # levels in the raster
nlevels(madCover) # number of categories
catNames(madCover) # names of categories table

plot(madCover)
```

`madForest2014` *Forest cover in year 2014 for a portion of Madagascar*

Description

Raster of occurrence/non-occurrence of forest cover in a portion of Madagascar. Cells are 30-m in resolution. Values represent forest (1) or non-forest (NA).

Format

An object of class `SpatRaster`..

References

Vielledent, G., Grinand, C., Rakotomala, F.A., Ranaivosoa, **R**., Rakotoarijaona, J-R., Allnutt, T.F., and Achard, F. 2018. Combining global tree cover loss data with historical national forest cover maps to look at six decades of deforestation and forest fragmentation in Madagascar. *Biological Conservation* 222:189-197. doi:10.1016/j.biocon.2018.04.008.

Examples

```
### vector data

library(sf)

# For vector data, we can use data(*) or fastData(*):
data(madCoast0) # same as next line
madCoast0 <- fastData("madCoast0") # same as previous
madCoast0
plot(st_geometry(madCoast0))

madCoast4 <- fastData("madCoast4")
madCoast4
plot(st_geometry(madCoast4), add = TRUE)

madRivers <- fastData("madRivers")
madRivers
plot(st_geometry(madRivers), col = "blue", add = TRUE)

madDypsis <- fastData("madDypsis")
madDypsis
plot(st_geometry(madDypsis), col = "red", add = TRUE)

### raster data

library(terra)

# For raster data, we can get the file directly or using fastData(*):
rastFile <- system.file("extdata/madElev.tif", package="fasterRaster")
madElev <- terra::rast(rastFile)

madElev <- fastData("madElev") # same as previous two lines
madElev
plot(madElev)

madForest2000 <- fastData("madForest2000")
madForest2000
plot(madForest2000)

madForest2014 <- fastData("madForest2014")
madForest2014
plot(madForest2014)

# multi-layer rasters
madChelsa <- fastData("madChelsa")
```

```
madChelsa
plot(madChelsa)

madPpt <- fastData("madPpt")
madTmin <- fastData("madTmin")
madTmax <- fastData("madTmax")
madPpt
madTmin
madTmax


# RGB raster
madLANDSAT <- fastData("madLANDSAT")
madLANDSAT
plotRGB(madLANDSAT, 4, 1, 2, stretch = "lin")

# categorical raster
madCover <- fastData("madCover")
madCover
madCover <- droplevels(madCover)
levels(madCover) # levels in the raster
nlevels(madCover) # number of categories
catNames(madCover) # names of categories table

plot(madCover)
```

`madLANDSAT` *Rasters of surface reflectance for an eastern portion of Madagascar*

Description

Raster layers of surface reflectance from LANDSAT 9 for an eastern portion of Madagascar taken May 21, 2023. Four bands are represented:

- 'band2': Blue (450-510 nm)
- 'band3': Green (530-590 nm)
- 'band4': Red (640-670 nm)
- 'band5': Near-infrared (850-880 nm) The rasters have been resampled to 90-m resolution to reduce their size, then rescaled to integers in the range 0 to 255.

Format

An object of class `SpatRaster` in Universal Trans-Mercator (UTM), Zone 39 North with a WGS84 coordinate system, at 90 m resolution.

Source

United States Geological Survey's EarthExplorer. Also see band definitions.

Examples

```
### vector data

library(sf)

# For vector data, we can use data(*) or fastData(*):
data(madCoast0) # same as next line
madCoast0 <- fastData("madCoast0") # same as previous
madCoast0
plot(st_geometry(madCoast0))

madCoast4 <- fastData("madCoast4")
madCoast4
plot(st_geometry(madCoast4), add = TRUE)

madRivers <- fastData("madRivers")
madRivers
plot(st_geometry(madRivers), col = "blue", add = TRUE)

madDypsis <- fastData("madDypsis")
madDypsis
plot(st_geometry(madDypsis), col = "red", add = TRUE)

### raster data

library(terra)

# For raster data, we can get the file directly or using fastData(*):
rastFile <- system.file("extdata/madElev.tif", package="fasterRaster")
madElev <- terra::rast(rastFile)

madElev <- fastData("madElev") # same as previous two lines
madElev
plot(madElev)

madForest2000 <- fastData("madForest2000")
madForest2000
plot(madForest2000)

madForest2014 <- fastData("madForest2014")
madForest2014
plot(madForest2014)

# multi-layer rasters
madChelsa <- fastData("madChelsa")
madChelsa
plot(madChelsa)

madPpt <- fastData("madPpt")
madTmin <- fastData("madTmin")
madTmax <- fastData("madTmax")
madPpt
```

```
madTmin
madTmax


# RGB raster
madLANDSAT <- fastData("madLANDSAT")
madLANDSAT
plotRGB(madLANDSAT, 4, 1, 2, stretch = "lin")

# categorical raster
madCover <- fastData("madCover")
madCover
madCover <- droplevels(madCover)
levels(madCover) # levels in the raster
nlevels(madCover) # number of categories
catNames(madCover) # names of categories table

plot(madCover)
```

`madPpt` *Rasters of average monthly precipitation for an eastern portion of Madagascar*

Description

Rasters of precipitation for an eastern portion of Madagascar from WorldClim 2.1 at ~3.33 arcminute resolution projected to the Tananarive (Paris)/Laborde Grid coordinate reference system. Values represent monthly averages across 1970-2000. Units are in millimeters. These should not be used for formal analysis.

Format

An object of class `SpatRaster`.

Source

doi:10.1002/joc.5086

References

Fick, S.E. and Hijmans, R.J. 2017. WorldClim 2: New 1-km spatial resolution climate surfaces for global land areas. International Journal of Climatology 37:4302-4315. doi:10.1002/joc.5086

Examples

```
### vector data

library(sf)
```

```
# For vector data, we can use data(*) or fastData(*):
data(madCoast0) # same as next line
madCoast0 <- fastData("madCoast0") # same as previous
madCoast0
plot(st_geometry(madCoast0))

madCoast4 <- fastData("madCoast4")
madCoast4
plot(st_geometry(madCoast4), add = TRUE)

madRivers <- fastData("madRivers")
madRivers
plot(st_geometry(madRivers), col = "blue", add = TRUE)

madDypsis <- fastData("madDypsis")
madDypsis
plot(st_geometry(madDypsis), col = "red", add = TRUE)

### raster data

library(terra)

# For raster data, we can get the file directly or using fastData(*):
rastFile <- system.file("extdata/madElev.tif", package="fasterRaster")
madElev <- terra::rast(rastFile)

madElev <- fastData("madElev") # same as previous two lines
madElev
plot(madElev)

madForest2000 <- fastData("madForest2000")
madForest2000
plot(madForest2000)

madForest2014 <- fastData("madForest2014")
madForest2014
plot(madForest2014)

# multi-layer rasters
madChelsa <- fastData("madChelsa")
madChelsa
plot(madChelsa)

madPpt <- fastData("madPpt")
madTmin <- fastData("madTmin")
madTmax <- fastData("madTmax")
madPpt
madTmin
madTmax


# RGB raster
madLANDSAT <- fastData("madLANDSAT")
```

```
madLANDSAT
plotRGB(madLANDSAT, 4, 1, 2, stretch = "lin")

# categorical raster
madCover <- fastData("madCover")
madCover
madCover <- droplevels(madCover)
levels(madCover) # levels in the raster
nlevels(madCover) # number of categories
catNames(madCover) # names of categories table

plot(madCover)
```

`madRivers` *Major rivers in a selected portion of Madagascar*

Description

Spatial lines object of major rivers in a portion of Madagascar.

Usage

```
data(madRivers)
```

Format

An object of class `sf`.

Source

DIVA-GIS

Examples

```
### vector data

library(sf)

# For vector data, we can use data(*) or fastData(*):
data(madCoast0) # same as next line
madCoast0 <- fastData("madCoast0") # same as previous
madCoast0
plot(st_geometry(madCoast0))

madCoast4 <- fastData("madCoast4")
madCoast4
plot(st_geometry(madCoast4), add = TRUE)

madRivers <- fastData("madRivers")
madRivers
```

```
plot(st_geometry(madRivers), col = "blue", add = TRUE)

madDypsis <- fastData("madDypsis")
madDypsis
plot(st_geometry(madDypsis), col = "red", add = TRUE)

### raster data

library(terra)

# For raster data, we can get the file directly or using fastData(*):
rastFile <- system.file("extdata/madElev.tif", package="fasterRaster")
madElev <- terra::rast(rastFile)

madElev <- fastData("madElev") # same as previous two lines
madElev
plot(madElev)

madForest2000 <- fastData("madForest2000")
madForest2000
plot(madForest2000)

madForest2014 <- fastData("madForest2014")
madForest2014
plot(madForest2014)

# multi-layer rasters
madChelsa <- fastData("madChelsa")
madChelsa
plot(madChelsa)

madPpt <- fastData("madPpt")
madTmin <- fastData("madTmin")
madTmax <- fastData("madTmax")
madPpt
madTmin
madTmax


# RGB raster
madLANDSAT <- fastData("madLANDSAT")
madLANDSAT
plotRGB(madLANDSAT, 4, 1, 2, stretch = "lin")

# categorical raster
madCover <- fastData("madCover")
madCover
madCover <- droplevels(madCover)
levels(madCover) # levels in the raster
nlevels(madCover) # number of categories
catNames(madCover) # names of categories table

plot(madCover)
```

madTmax *Rasters of average monthly maximum temperature for an eastern portion of Madagascar*

Description

Rasters of maximum temperature for an eastern portion of Madagascar from WorldClim 2.1 at ~3.3 arcminute resolution projected to the Tananarive (Paris)/Laborde Grid coordinate reference system. Values represent monthly averages across 1970-2000. Units are in degrees C. These should not be used for formal analysis.

Format

An object of class `SpatRaster`.

Source

doi:10.1002/joc.5086

References

Fick, S.E. and Hijmans, R.J. 2017. WorldClim 2: New 1-km spatial resolution climate surfaces for global land areas. International Journal of Climatology 37:4302-4315. doi:10.1002/joc.5086

Examples

```
### vector data

library(sf)

# For vector data, we can use data(*) or fastData(*):
data(madCoast0) # same as next line
madCoast0 <- fastData("madCoast0") # same as previous
madCoast0
plot(st_geometry(madCoast0))

madCoast4 <- fastData("madCoast4")
madCoast4
plot(st_geometry(madCoast4), add = TRUE)

madRivers <- fastData("madRivers")
madRivers
plot(st_geometry(madRivers), col = "blue", add = TRUE)

madDypsis <- fastData("madDypsis")
madDypsis
plot(st_geometry(madDypsis), col = "red", add = TRUE)

### raster data
```

```
library(terra)

# For raster data, we can get the file directly or using fastData(*):
rastFile <- system.file("extdata/madElev.tif", package="fasterRaster")
madElev <- terra::rast(rastFile)

madElev <- fastData("madElev") # same as previous two lines
madElev
plot(madElev)

madForest2000 <- fastData("madForest2000")
madForest2000
plot(madForest2000)

madForest2014 <- fastData("madForest2014")
madForest2014
plot(madForest2014)

# multi-layer rasters
madChelsa <- fastData("madChelsa")
madChelsa
plot(madChelsa)

madPpt <- fastData("madPpt")
madTmin <- fastData("madTmin")
madTmax <- fastData("madTmax")
madPpt
madTmin
madTmax


# RGB raster
madLANDSAT <- fastData("madLANDSAT")
madLANDSAT
plotRGB(madLANDSAT, 4, 1, 2, stretch = "lin")

# categorical raster
madCover <- fastData("madCover")
madCover
madCover <- droplevels(madCover)
levels(madCover) # levels in the raster
nlevels(madCover) # number of categories
catNames(madCover) # names of categories table

plot(madCover)
```

---

madTmin *Rasters of average monthly minimum temperature for an eastern portion of Madagascar*

Description

Rasters of minimum temperature for an eastern portion of Madagascar from WorldClim 2.1 at ~3.33 arcminute resolution projected to the Tananarive (Paris)/Laborde Grid coordinate reference system. Values represent monthly averages across 1970-2000. Units are in degrees C. These should not be used for formal analysis.

Format

An object of class `SpatRaster`.

Source

doi:10.1002/joc.5086

References

Fick, S.E. and Hijmans, R.J. 2017. WorldClim 2: New 1-km spatial resolution climate surfaces for global land areas. International Journal of Climatology 37:4302-4315. doi:10.1002/joc.5086

Examples

```
### vector data

library(sf)

# For vector data, we can use data(*) or fastData(*):
data(madCoast0) # same as next line
madCoast0 <- fastData("madCoast0") # same as previous
madCoast0
plot(st_geometry(madCoast0))

madCoast4 <- fastData("madCoast4")
madCoast4
plot(st_geometry(madCoast4), add = TRUE)

madRivers <- fastData("madRivers")
madRivers
plot(st_geometry(madRivers), col = "blue", add = TRUE)

madDypsis <- fastData("madDypsis")
madDypsis
plot(st_geometry(madDypsis), col = "red", add = TRUE)

### raster data

library(terra)

# For raster data, we can get the file directly or using fastData(*):
rastFile <- system.file("extdata/madElev.tif", package="fasterRaster")
madElev <- terra::rast(rastFile)
```

```
madElev <- fastData("madElev") # same as previous two lines
madElev
plot(madElev)

madForest2000 <- fastData("madForest2000")
madForest2000
plot(madForest2000)

madForest2014 <- fastData("madForest2014")
madForest2014
plot(madForest2014)

# multi-layer rasters
madChelsa <- fastData("madChelsa")
madChelsa
plot(madChelsa)

madPpt <- fastData("madPpt")
madTmin <- fastData("madTmin")
madTmax <- fastData("madTmax")
madPpt
madTmin
madTmax


# RGB raster
madLANDSAT <- fastData("madLANDSAT")
madLANDSAT
plotRGB(madLANDSAT, 4, 1, 2, stretch = "lin")

# categorical raster
madCover <- fastData("madCover")
madCover
madCover <- droplevels(madCover)
levels(madCover) # levels in the raster
nlevels(madCover) # number of categories
catNames(madCover) # names of categories table

plot(madCover)
```

`mask,GRaster,GRaster-method`

Mask values in a raster

Description

The output of `mask()` is a `GRaster` that has the same as values as the input raster. However, if the `mask` argument is a `GRaster`, the output will have `NA` values in the same cells that the `mask` raster has `NA` cells. If the `mask` argument is a `GVector`, then the output raster will have `NA` values in cells the `GVector` does not cover.

Usage

```
## S4 method for signature 'GRaster,GRaster'
mask(x, mask, inverse = FALSE, maskvalues = NA, updatevalue = NA)

## S4 method for signature 'GRaster,GVector'
mask(x, mask, inverse = FALSE, updatevalue = NA)
```

Arguments

x	A GRaster.
mask	A GRaster or GVector.
inverse	Logical: If TRUE, the effect of the mask is inverted. That is, a copy of the input raster is made, but cells that overlap with an NA in the mask raster or are not covered by the mask vector retain their values. Cells that overlap with an NA in the mask raster or overlap with the mask vector are forced to NA.
maskvalues	Numeric vector, including NA (only for when mask is a GRaster): The value(s) in the mask raster cells that serve as the mask. The default is NA, in which case cells in the input raster that overlap with NA cells in the mask are forced to NA.
updatevalue	Numeric, including NA (default): The values assigned to masked cells.

Value

A GRaster.

See Also

terra::mask(), **GRASS** module r.mask

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Example data
madElev <- fastData("madElev") # raster
madForest <- fastData("madForest2000") # raster
madCoast <- fastData("madCoast4") # vector

# Convert to GRasters and GVectors
elev <- fast(madElev)
forest <- fast(madForest)
coast <- fast(madCoast)

ant <- coast[coast$NAME_4 == "Antanambe"]

# Mask by a raster or  vector:
```

```
maskByRast <- mask(elev, forest)
plot(c(forest, maskByRast))

maskByVect <- mask(elev, ant)
plot(maskByVect)
plot(ant, add = TRUE)

# Mask by a raster or vector, but invert mask:
maskByRastInvert <- mask(elev, forest, inverse = TRUE)
plot(c(forest, maskByRastInvert))

maskByVectInvert <- mask(elev, ant, inverse = TRUE)
plot(maskByVectInvert)
plot(ant, add = TRUE)

# Mask by a raster, but use custom values for the mask:
maskByRastCustomMask <- mask(elev, elev, maskvalues = 1:20)
plot(c(elev <= 20, maskByRastCustomMask))

# Mask by a raster or vector, but force masked values to a custom value:
byRastCustomUpdate <- mask(elev, forest, updatevalue = 7)
plot(byRastCustomUpdate)

byVectCustomUpdate <- mask(elev, ant, updatevalue = 7)
plot(byVectCustomUpdate)

# Mask by a raster, inverse, custom values, and custom update:
byRastAll <-
   mask(elev, elev, inverse = TRUE, maskvalues = 1:20, updatevalue = 7)

plot(c(elev, byRastAll))

}
```

---

maskNA,GRaster-method *Mask all non-NA cells or all NA cells*

Description

This function converts all non-NA cells in a GRaster to a single user-defined value, leaving NA cells as NA. Alternatively, it can convert NA cells to a user-defined value, and all non-NA cells to NA.

Usage

```
## S4 method for signature 'GRaster'
maskNA(x, value = 1, invert = FALSE, retain = FALSE)
```

Arguments

x	A GRaster.
value	Numeric: Value to which to assign to masked cells. The default is 1.
invert	Logical: If FALSE (default), convert non-NA cells to value, and leave NA cells as-is. If TRUE, convert all NA cells to value, and non-NA cells to NA.
retain	Logical: If invert is TRUE and retain is FALSE (default), non-NA cells will retain their value. This argument is ignored if invert is FALSE.

Value

A GRaster.

See Also

not.na(), app(), mask()

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Elevation raster
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster:
elev <- fast(madElev)

# Convert non-NA to 1, NA cells remain NA
elevMask <- maskNA(elev)
elevMask
plot(c(elev, elevMask))

# Convert NA to 1, non-NA cells become NA
elevInvertMask <- maskNA(elev, invert = TRUE)
elevInvertMask
plot(c(elev, elevInvertMask))

# Convert NA to 200, non-NA cells keep their values
elevInvertRetain <- maskNA(elev, value = 200, invert = TRUE, retain = TRUE)
elevInvertRetain
plot(c(elev, elevInvertRetain))

}
```

`match,GRaster-method` *Find which cells of a GRaster match certain values*

Description

The `match()` function takes a `GRaster` and a numeric, integer or character vector as inputs and returns a `GRaster` with cell values that correspond to the index of each element in the vector that matched the original cell value. For example, if a 4-cell raster had values 3, `NA`, 5, 4, and the vector was `c(3, 4)`, then the output would be a 4-cell raster with values 1, `NA`, `NA`, 2 because the first value in the vector was 3 (so the cell with 3 is assigned 1), and because the second value in the vector was 4 (so the cell with 4 was assigned 2). The other two values had no matches.

If the `GRaster` is categorical, then the vector can be category labels instead of numeric values.

The `%in%` operator returns a `GRaster` with cell values that are 1 if their original values appeared in the vector, and 0 if not (or `NA` if the original value was `NA`). If the `GRaster` is categorical, then the vector can be category labels instead of numeric values.

The `%notin%` operator returns 1 for cells with values that are *not* found in the vector, and 0 otherwise. If the `GRaster` is categorical, then the vector can be category labels instead of numeric values.

Usage

```
## S4 method for signature 'GRaster'
match(x, table, nomatch = NA)

## S4 method for signature 'GRaster'
x %in% table

## S4 method for signature 'GRaster'
x %notin% table
```

Arguments

`x`	A `GRaster`: Note that any kind of `GRaster` is acceptable (integer, float, double, or categorical), but matching may not work as intended for float and double rasters because of problems with comparing floating-point values.
`table`	A numeric, integer, or character vector.
`nomatch`	Numeric or integer: Value to return when no match is found.

Value

A `GRaster`.

See Also

`terra::match()`, `match()`, `omnibus::notIn()`

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data: Elevation and land cover rasters
madElev <- fastData("madElev")
madCover <- fastData("madCover")

### match() with an integer raster:

elev <- fast(madElev)

# Cells in elevation raster replaced with index in which they appear
# in the table:
table <- c(10, 20, 30, 40, 50)
elevIndex <- match(elev, table)
elevIndexNeg <- match(elev, table, nomatch = -100)

plot(c(elevIndex, elevIndexNeg))

### Using %in% and %notin% on an integer GRaster:

elev <- fast(madElev)
table <- c(10, 20, 30, 40, 50)

ins <- elev %in% table
notins <- elev %notin% table

plot(c(ins, notins))

### match() with a categorical raster:

cover <- fast(madCover)
cover <- droplevels(cover)
levels(cover)

forestLabels <- c(
   "Sparse broadleaved evergreen/semi-deciduous forest",
   "Broadleaved deciduous forest",
   "Grassland with mosaic forest",
   "Flooded forest"
)

forestClasses <- match(cover, forestLabels)
plot(forestClasses)
levels(forestClasses)

forestNoMatch <- match(cover, forestLabels, nomatch = -100)
plot(forestNoMatch)
levels(forestNoMatch)
```

```
### Using %in% and %notin% on a categorical GRaster:

cover <- fast(madCover)
cover <- droplevels(cover)
levels(cover)

forestLabels <- c(
   "Sparse broadleaved evergreen/semi-deciduous forest",
   "Broadleaved deciduous forest",
   "Grassland with mosaic forest",
   "Flooded forest"
)

forest <- cover %in% forestLabels
plot(forest)

notForest <- cover %notin% forestLabels
plot(notForest)

}
```

`mean,GRaster-method` *Mathematical operations on two or more GRasters*

Description

These functions can be applied to a "stack" of `GRasters` with two or more layers. They return a single-layered `GRaster`. If you want to summarize across cells in a raster (e.g., calculate the mean value of all cells on a raster), use `global()`. Options include:

- Numeration: `count()` (number of non-`NA` cells), `sum()`.
- Central tendency: `mean()`, `mmode()` (mode), `median()`.
- Extremes: `min()`, `max()`, `which.min()` (index of raster with the minimum value), `which.max()` (index of the raster with the maximum value)
- Dispersion: `range()`, `stdev()` (standard deviation), `var()` (sample variance), `varpop()` (population variance), `nunique()` (number of unique values), `quantile()` (use argument `probs`), `skewness()`, and `kurtosis()`.
- `NA`s: `anyNA()` (any cells are `NA`?), `allNA()` (are all cells `NA`?)

Usage

```
## S4 method for signature 'GRaster'
mean(x, na.rm = FALSE)

## S4 method for signature 'GRaster'
mmode(x, na.rm = FALSE)
```

```
## S4 method for signature 'GRaster'
median(x, na.rm = FALSE)

## S4 method for signature 'GRaster'
count(x)

## S4 method for signature 'GRaster'
sum(x, na.rm = FALSE)

## S4 method for signature 'GRaster'
min(x, na.rm = FALSE)

## S4 method for signature 'GRaster'
max(x, na.rm = FALSE)

## S4 method for signature 'GRaster'
which.min(x)

## S4 method for signature 'GRaster'
which.max(x)

## S4 method for signature 'numeric'
sdpop(x, na.rm = FALSE)

## S4 method for signature 'GRaster'
varpop(x, na.rm = FALSE)

## S4 method for signature 'numeric'
varpop(x, na.rm = FALSE)

## S4 method for signature 'GRaster'
stdev(x, pop = TRUE, na.rm = FALSE)

## S4 method for signature 'GRaster'
var(x, na.rm = FALSE)

## S4 method for signature 'GRaster'
nunique(x, na.rm = FALSE)

## S4 method for signature 'GRaster'
skewness(x, na.rm = FALSE)

## S4 method for signature 'GRaster'
kurtosis(x, na.rm = FALSE)

## S4 method for signature 'GRaster'
range(x, na.rm = FALSE)
```

```
## S4 method for signature 'GRaster'
quantile(x, prob, na.rm = FALSE)

## S4 method for signature 'GRaster'
anyNA(x)

## S4 method for signature 'GRaster'
allNA(x)
```

Arguments

x	A GRaster. Typically, this raster will have two or more layers. Values will be calculated within cells across rasters.
na.rm	Logical: If FALSE (default), of one cell value has an NA, the result will be NA. If TRUE, NAs are ignored.
pop	Logical (for stdev()): If TRUE (default), calculate the population standard deviation across layers. If FALSE, calculate the sample standard deviation.
prob	Numeric: Quantile to calculate. Used for quantile().

Value

A GRaster.

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Example data
madChelsa <- fastData("madChelsa")

# Convert a SpatRaster to a GRaster
chelsa <- fast(madChelsa)
chelsa # 4 layers

# Central tendency
mean(chelsa)
mmode(chelsa)
median(chelsa)

# Statistics
nunique(chelsa)
sum(chelsa)
count(chelsa)
min(chelsa)
max(chelsa)
range(chelsa)
```

```
skewness(chelsa)
kurtosis(chelsa)

stdev(chelsa)
stdev(chelsa, pop = FALSE)
var(chelsa)
varpop(chelsa)

# Which layers have maximum/minimum?
which.min(chelsa)
which.max(chelsa)

# Regression

# Note the intercept is different for fasterRaster::regress().
regress(chelsa)
regress(madChelsa, 1:nlyr(madChelsa))

# Note: To get quantiles for each layer, use global().
quantile(chelsa, 0.1)

# NAs
madForest2000 <- fastData("madForest2000")
forest2000 <- fast(madForest2000)
forest2000 <- project(forest2000, chelsa, method = "near")

chelsaForest <- c(chelsa, forest2000)

nas <- anyNA(chelsaForest)
plot(nas)

allNas <- allNA(chelsaForest)
plot(allNas)

}
```

`merge,GRaster,GRaster-method`
Combine two or more rasters with different extents and fill in NAs

---

Description

`merge()` combines two or more `GRaster`s, possibly with different extents, into a single larger `GRaster`. Where the same cell has different values in each raster, the value of the first raster's cell is used. If this is `NA`, then the value of the second raster's cell is used, and so on.

Usage

```
## S4 method for signature 'GRaster,GRaster'
merge(x, y, ...)
```

Arguments

x, y, ...	GRasters.

Value

A GRaster.

See Also

terra::merge(), terra::mosaic(), and **GRASS** module r.patch

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Example data
madElev <- fastData("madElev")
madCoast4 <- fastData("madCoast4")
madCoast4 <- vect(madCoast4)

# For the example, crop the elevation raster to two communes
madAnt <- madCoast4[madCoast4$NAME_4 == "Antanambe", ]
madMan <- madCoast4[madCoast4$NAME_4 == "Manompana", ]

elevAnt <- crop(madElev, madAnt)
elevMan <- crop(madElev, madMan)

plot(madElev)
plot(elevAnt, col = "red", legend = FALSE, add = TRUE)
plot(elevMan, col = "blue", legend = FALSE, add = TRUE)

# Convert a SpatRaster to a GRaster
ant <- fast(elevAnt)
man <- fast(elevMan)

# merge
antMan <- merge(ant, man)
plot(antMan, main = "Antman!")

}
```

---

minmax,GRaster-method *Minimum and maximum values or categories of a GRaster*

Description

`minmax()` reports the minimum and maximum values across all non-NA cells of a `GRaster`. When the `levels` argument is `TRUE` and the raster is categorical, the function reports the "lowest" and "highest" category values in a categorical (factor) `GRaster`.

Usage

```
## S4 method for signature 'GRaster'
minmax(x, levels = FALSE)
```

Arguments

`x`	A `GRaster`.
`levels`	Logical: If `TRUE` and the raster is a categorical raster, return the "lowest" and "highest" categories. The default is `FALSE`.

Value

`minmax()` returns a numeric matrix, and `minmax(..., levels = TRUE)` returns a `data.frame` with category names. In the latter case, non-categorical rasters will have `NA` values.

See Also

`terra::minmax()`

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")
madForest2000 <- fastData("madForest2000")

# Convert SpatRasters to GRasters
elev <- fast(madElev)
forest <- fast(madForest2000)

### GRaster properties

# plotting
plot(elev)

# dimensions
dim(elev) # rows, columns, depths, layers
nrow(elev) # rows
ncol(elev) # columns
ndepth(elev) # depths
nlyr(elev) # layers
```

```
res(elev) # resolution (2D)
res3d(elev) # resolution (3D)
zres(elev) # vertical resolution
xres(elev) # vertical resolution
yres(elev) # vertical resolution
zres(elev) # vertical resolution (NA because this is a 2D GRaster)

# cell counts
ncell(elev) # cells
ncell3d(elev) # cells (3D rasters only)

# number of NA and non-NA cells
nacell(elev)
nonnacell(elev)

# topology
topology(elev) # number of dimensions
is.2d(elev) # is it 2-dimensional?
is.3d(elev) # is it 3-dimensional?

minmax(elev) # min/max values

# "names" of the object
names(elev)

# coordinate reference system
crs(elev)
st_crs(elev)
coordRef(elev)

# extent (bounding box)
ext(elev)

# vertical extent (not defined for this raster)
zext(elev)

# data type
datatype(elev) # fasterRaster type
datatype(elev, "GRASS") # GRASS type
datatype(elev, "terra") # terra type
datatype(elev, "GDAL") # GDAL type

is.integer(elev)
is.float(elev)
is.double(elev)
is.factor(elev)

# convert data type
as.int(elev) # integer; note that "elev" is already of type "integer"
as.float(elev) # floating-precision
as.doub(elev) # double-precision
```

```
# assigning
pie <- elev
pie[] <- pi # assign all cells to the value of pi
pie

# concatenating multiple GRasters
rasts <- c(elev, forest)
rasts

# subsetting
rasts[[1]]
rasts[["madForest2000"]]

# replacing
rasts[[2]] <- 2 * forest
rasts

# adding layers
rasts[[3]] <- elev > 500 # add a layer
rasts <- c(rasts, sqrt(elev)) # add another
add(rasts) <- ln(elev)
rasts

# names
names(rasts)
names(rasts) <- c("elev_meters", "2_x_forest", "high_elevation", "sqrt_elev", "ln_elev")
rasts

# remove a layer
rasts[["2_x_forest"]] <- NULL
rasts

# number of layers
nlyr(rasts)

# correlation and covariance matrices
madLANDSAT <- fastData("madLANDSAT")
landsat <- fast(madLANDSAT) # projects matrix
layerCor(landsat) # correlation
layerCor(landsat, fun = 'cov') # covariance

}
```

---

missingCats,GRaster-method

Values in a categorical raster with no assigned category

Description

This function reports the values in a categorical `GRaster` that have no matching category label in its "levels" table.

GRasters can represent categorical data. Cell values are actually integers, each corresponding to a category, such as "desert" or "wetland." A categorical raster is associated with a table that matches each value to a category name.

Usage

```
## S4 method for signature 'GRaster'
missingCats(x, layer = 1:nlyr(x))
```

Arguments

x	A GRaster.
layer	Numeric integers, logical vector, or character: Layer(s) for which to obtain missing categories.

Value

A numeric vector (if xis just one layer), or a named list of numeric vectors, one per layer in x.

See Also

missingCats(), missing.cases(), droplevels(), vignette("GRasters", package = "fasterRaster")

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data: Land cover raster
madCover <- fastData("madCover")

# Convert categorical SpatRaster to categorical GRaster:
cover <- fast(madCover)

### Properties of categorical rasters

cover # note categories
is.factor(cover) # Is the raster categorical?
nlevels(cover) # number of levels
levels(cover) # just the value and active column
cats(cover) # all columns
minmax(cover) # min/max values
minmax(cover, levels = TRUE) # min/max categories
catNames(cover) # column names of the levels table
missingCats(cover) # categories in table with no values in raster
freq(cover) # frequency of each category (number of cells)
zonalGeog(cover) # geometric statistics

### Active column
```

```
# Which column sets the category labels?
activeCat(cover)
activeCat(cover, names = TRUE)

activeCats(c(cover, cover))

# Choose a different column for category labels:
levels(cover)
activeCat(cover) <- 2
levels(cover)

### Managing levels tables

# Remove unused levels:
nlevels(cover)
cover <- droplevels(cover)
nlevels(cover)

# Re-assign levels:
value <- c(20, 30, 40, 50, 120, 130, 140, 170)
label <- c("Cropland", "Cropland", "Forest", "Forest",
 "Grassland", "Shrubland", "Herbaceous", "Flooded")

newCats <- data.frame(value = value, label = label)

cover <- categories(cover, layer = 1, value = newCats)
cats(cover)

# This is the same as:
levels(cover) <- newCats
cats(cover)

# Are there any values not assigned a category?
missingCats(cover)

# Let's assign a category for value 210 (water):
water <- data.frame(value = 210, label = "Water")
addCats(cover) <- water
levels(cover)

# Add more information to the levels table using merge():
landType <- data.frame(
     Value = c(20, 30, 40, 50, 120),
     Type = c("Irrigated", "Rainfed", "Broadleaf evergreen",
     "Broadleaf deciduous", "Mosaic with forest")
)
cats(cover)
cover <- addCats(cover, landType, merge = TRUE)
cats(cover)

### Logical operations on categorical rasters
```

```
cover < "Forest" # 1 for cells with a value < 40, 0 otherwise
cover <= "Forest" # 1 for cells with a value < 120, 0 otherwise
cover == "Forest" # 1 for cells with value of 40-120, 0 otherwise
cover != "Forest" # 1 for cells with value that is not 40-120, 0 otherwise
cover > "Forest" # 1 for cells with a value > 120, 0 otherwise
cover >= "Forest" # 1 for cells with a value >= 120, 0 otherwise

cover %in% c("Cropland", "Forest") # 1 for cropland/forest cells, 0 otherwise

### Combine categories from different rasters

# For the example, will create a second categorical raster fromm elevation.

# Divide elevation raster into "low/medium/high" levels:
madElev <- fastData("madElev")
elev <- fast(madElev)
elev <- project(elev, cover, method = "near") # convert to same CRS
fun <- "= if(madElev < 100, 0, if(madElev < 400, 1, 2))"
elevCat <- app(elev, fun)

levs <- data.frame(
     value = c(0, 1, 2),
     elevation = c("low", "medium", "high")
)
levels(elevCat) <- list(levs)

# Combine levels:
combined <- concats(cover, elevCat)
combined
levels(combined)

# Combine levels, treating value/NA combinations as new categories:
combinedNA <- concats(cover, elevCat, na.rm = FALSE)
combinedNA
levels(combinedNA)

}
```

---

mow *Remove rasters and vectors from the GRASS cache*

Description

fasterRaster functions attempt to delete rasters and vectors in the **GRASS** cache, but not all intermediate files can be removed. This function can be used to clear the cache of extraneous rasters and vectors.

Calling this function inside another function's environment and defining `x` as `"*"` can be very **dangerous**, as it will detect objects outside of that environment, and thus delete any rasters/vectors outside that environment. Here is a guide:

- To delete files associated with a single GRaster or GVector, use mow(GRaster_to_unlink) or mow(GVector_to_unlink). To remove all rasters, all vectors, or all rasters and vectors in the **GRASS** cache that are not linked to a GRaster or GVector, use mow("*"). To remove all rasters or all vectors in the **GRASS** cache, use mow("*", type = "rasters") or mow("*", type = "vectors"). To remove all rasters or all vectors in the **GRASS** cache *except* for certain ones, use mow("*", unlinked = FALSE, keep = list(GRaster_to_keep, GVector_to_keep)). You can combine this with the keep argument to retain specific rasters or vectors. For example, you can use mow("*", unlinked = FALSE, type = "rasters", keep = list(GRaster_to_keep)).

Usage

```
mow(
  x = "unlinked",
  pos = NULL,
  type = NULL,
  keep = NULL,
  verbose = TRUE,
  ask = TRUE
)
```

Arguments

x	Any of: • "unlinked" (default): Delete **GRASS** rasters and/or vectors that are unlinked to GRasters or GVectors in the environment in which the function was called, or the environment named in pos. • A GRaster or GVector: Delete the **GRASS** raster or vector pointed to by this object. • A list of GRasters and/or GVectors: Delete the **GRASS** raster(s) and/or vector(s) pointed to by these objects. • "*": Delete *all* **GRASS** rasters and/or vectors pointed to by objects in the environment named in pos. Only objects in keep will not be deleted.
pos	Either NULL (default), or an environment. This is used only if x is "unlinked" or "*". In that case, if pos is NULL, the environment in which this function was called will be searched for GRasters and GVectors for removal of their associated **GRASS** rasters and vectors. Otherwise, the named environment will be searched.
type	Either NULL or a character vector. This is used only if x is "unlinked" or "*". If NULL, all rasters and vectors in the **GRASS** cache are candidates for deletion. Otherwise, this can be either "rasters", "vectors", or both.
keep	Either NULL (default) or a list() of GRasters and/or GVectors that you want to retain. This is used only if x is "unlinked" or "*". The rasters and vectors in **GRASS** pointed to by these objects will not be deleted.
verbose	Logical: If TRUE (default), report progress.
ask	Logical: If TRUE (default), prompt for reassurance. This is used only if x is "unlinked" or "*".

Value

Invisibly returns a named vector with the number of rasters and vectors deleted.

See Also

terra::tmpFiles()

Examples

```
if (grassStarted()) {

# Setup
madElev <- fastData("madElev")
elev <- fast(madElev)

mow(elev, ask = TRUE) # delete GRASS raster attached to `elev`

}
```

---

nacell,GRaster-method *Number of NA or non-NA cells in a raster*

Description

The nacell() function counts the number of NA cells in a GRaster, and the nonnacell() reports the number of non-NA cells. If the raster is 3D, then all cells in all layers are counted.

Usage

```
## S4 method for signature 'GRaster'
nacell(x, warn = TRUE)

## S4 method for signature 'GRaster'
nonnacell(x, warn = TRUE)
```

Arguments

x	A GRaster.
warn	Logical: If TRUE (default), display a warning about how much time the computation could take.

Value

A numeric value, one per raster layer in the input.

See Also

ncell(), ncell3d(), terra::ncell(), dim(), terra::dim()

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")
madForest2000 <- fastData("madForest2000")

# Convert SpatRasters to GRasters
elev <- fast(madElev)
forest <- fast(madForest2000)

### GRaster properties

# plotting
plot(elev)

# dimensions
dim(elev) # rows, columns, depths, layers
nrow(elev) # rows
ncol(elev) # columns
ndepth(elev) # depths
nlyr(elev) # layers

res(elev) # resolution (2D)
res3d(elev) # resolution (3D)
zres(elev) # vertical resolution
xres(elev) # vertical resolution
yres(elev) # vertical resolution
zres(elev) # vertical resolution (NA because this is a 2D GRaster)

# cell counts
ncell(elev) # cells
ncell3d(elev) # cells (3D rasters only)

# number of NA and non-NA cells
nacell(elev)
nonnacell(elev)

# topology
topology(elev) # number of dimensions
is.2d(elev) # is it 2-dimensional?
is.3d(elev) # is it 3-dimensional?

minmax(elev) # min/max values

# "names" of the object
names(elev)

# coordinate reference system
```

```
crs(elev)
st_crs(elev)
coordRef(elev)

# extent (bounding box)
ext(elev)

# vertical extent (not defined for this raster)
zext(elev)

# data type
datatype(elev) # fasterRaster type
datatype(elev, "GRASS") # GRASS type
datatype(elev, "terra") # terra type
datatype(elev, "GDAL") # GDAL type

is.integer(elev)
is.float(elev)
is.double(elev)
is.factor(elev)

# convert data type
as.int(elev) # integer; note that "elev" is already of type "integer"
as.float(elev) # floating-precision
as.doub(elev) # double-precision

# assigning
pie <- elev
pie[] <- pi # assign all cells to the value of pi
pie

# concatenating multiple GRasters
rasts <- c(elev, forest)
rasts

# subsetting
rasts[[1]]
rasts[["madForest2000"]]

# replacing
rasts[[2]] <- 2 * forest
rasts

# adding layers
rasts[[3]] <- elev > 500 # add a layer
rasts <- c(rasts, sqrt(elev)) # add another
add(rasts) <- ln(elev)
rasts

# names
names(rasts)
names(rasts) <- c("elev_meters", "2_x_forest", "high_elevation", "sqrt_elev", "ln_elev")
rasts
```

```
# remove a layer
rasts[["2_x_forest"]] <- NULL
rasts

# number of layers
nlyr(rasts)

# correlation and covariance matrices
madLANDSAT <- fastData("madLANDSAT")
landsat <- fast(madLANDSAT) # projects matrix
layerCor(landsat) # correlation
layerCor(landsat, fun = 'cov') # covariance

}
```

`names,GRaster-method` *Name(s) of a GRaster or columns of a GVector's data table*

Description

`names()` returns that names(s) of a `GRaster` or of columns of a `GVector`'s data table'.

Usage

```
## S4 method for signature 'GRaster'
names(x)

## S4 replacement method for signature 'GRaster'
names(x) <- value

## S4 method for signature 'GVector'
names(x)

## S4 replacement method for signature 'GVector'
names(x) <- value
```

Arguments

`x`	A `GRaster` or `GVector`.
`value`	Character: Name(s) to assign to the raster(s).

Details

`names(value) <-` assigns a new name to the `GRaster` or to the columns of a `GVector`'s data table.

Value

Character vector.

See Also

terra::names()

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")
madForest2000 <- fastData("madForest2000")

# Convert SpatRasters to GRasters
elev <- fast(madElev)
forest <- fast(madForest2000)

### GRaster properties

# plotting
plot(elev)

# dimensions
dim(elev) # rows, columns, depths, layers
nrow(elev) # rows
ncol(elev) # columns
ndepth(elev) # depths
nlyr(elev) # layers

res(elev) # resolution (2D)
res3d(elev) # resolution (3D)
zres(elev) # vertical resolution
xres(elev) # vertical resolution
yres(elev) # vertical resolution
zres(elev) # vertical resolution (NA because this is a 2D GRaster)

# cell counts
ncell(elev) # cells
ncell3d(elev) # cells (3D rasters only)

# number of NA and non-NA cells
nacell(elev)
nonnacell(elev)

# topology
topology(elev) # number of dimensions
is.2d(elev) # is it 2-dimensional?
is.3d(elev) # is it 3-dimensional?

minmax(elev) # min/max values
```

```
# "names" of the object
names(elev)

# coordinate reference system
crs(elev)
st_crs(elev)
coordRef(elev)

# extent (bounding box)
ext(elev)

# vertical extent (not defined for this raster)
zext(elev)

# data type
datatype(elev) # fasterRaster type
datatype(elev, "GRASS") # GRASS type
datatype(elev, "terra") # terra type
datatype(elev, "GDAL") # GDAL type

is.integer(elev)
is.float(elev)
is.double(elev)
is.factor(elev)

# convert data type
as.int(elev) # integer; note that "elev" is already of type "integer"
as.float(elev) # floating-precision
as.doub(elev) # double-precision

# assigning
pie <- elev
pie[] <- pi # assign all cells to the value of pi
pie

# concatenating multiple GRasters
rasts <- c(elev, forest)
rasts

# subsetting
rasts[[1]]
rasts[["madForest2000"]]

# replacing
rasts[[2]] <- 2 * forest
rasts

# adding layers
rasts[[3]] <- elev > 500 # add a layer
rasts <- c(rasts, sqrt(elev)) # add another
add(rasts) <- ln(elev)
rasts
```

```
# names
names(rasts)
names(rasts) <- c("elev_meters", "2_x_forest", "high_elevation", "sqrt_elev", "ln_elev")
rasts

# remove a layer
rasts[["2_x_forest"]] <- NULL
rasts

# number of layers
nlyr(rasts)

# correlation and covariance matrices
madLANDSAT <- fastData("madLANDSAT")
landsat <- fast(madLANDSAT) # projects matrix
layerCor(landsat) # correlation
layerCor(landsat, fun = 'cov') # covariance

}
```

---

ngeom,GVector-method *Number of geometries and subgeometries in a vector*

Description

GVectors represent two types of "geometries". In "singlepart" geometries, each point, set of connected line segments, or polygon is treated like its own feature and has its own row in an attribute table. For example, a province might be composed of islands. In this case, each island would be represented as its own feature and could have its own row in the attribute indicating, say, the name and area of each island.

In "multipart" geometries, features are collected together and thu manipulated as if they were a single feature and have a singe line in an attribute table. Each multipart feature can contain one or more singlepart features. For example, all of the islands comprising province would be collated together and have a single row in the attribute table indicating the name of the province and the area of the entire province.

ngeom() returns the number of geometries. Singlepart features are treated as one geometry each, and multipart features are treated as one geometry each.

nsubgeom() Returns the number of subgeometries. Singlepart geometries each represent a single subgeometry. Multipart geometries represent one or more subgeometries. The number of subgeometries will thus always be the same as or more than the number of geometries.

Usage

```
## S4 method for signature 'GVector'
ngeom(x)

## S4 method for signature 'GVector'
nsubgeom(x)
```

Arguments

x	A GVector.

Value

An integer.

See Also

nrow(), dim()

Examples

```
if (grassStarted()) {

# Setup
library(sf)

# Example data:
madCoast4 <- fastData("madCoast4")
madRivers <- fastData("madRivers")
madDypsis <- fastData("madDypsis")

# Convert sf vectors to GVectors:
coast <- fast(madCoast4)
rivers <- fast(madRivers)
dypsis <- fast(madDypsis)

# Geographic properties:
ext(rivers) # extent
crs(rivers) # coordinate reference system
st_crs(rivers) # coordinate reference system
coordRef(rivers) # coordinate reference system

# Column names and data types:
names(coast)
datatype(coast)

# Points, lines, or polygons?
geomtype(dypsis)
geomtype(rivers)
geomtype(coast)

is.points(dypsis)
is.points(coast)

is.lines(rivers)
is.lines(dypsis)

is.polygons(coast)
is.polygons(dypsis)
```

```
# Number of dimensions:
topology(rivers)
is.2d(rivers) # 2-dimensional?
is.3d(rivers) # 3-dimensional?

# Just the data table:
as.data.frame(rivers)
as.data.table(rivers)

# Top/bottom of the data table:
head(rivers)
tail(rivers)

# Vector or table with just selected columns:
names(rivers)
rivers$NAME
rivers[[c("NAM", "NAME_0")]]
rivers[[c(3, 5)]]

# Select geometries/rows of the vector:
nrow(rivers)
selected <- rivers[2:6]
nrow(selected)

# Plot:
plot(coast)
plot(rivers, col = "blue", add = TRUE)
plot(selected, col = "red", lwd = 2, add = TRUE)

# Vector math:
hull <- convHull(dypsis)

un <- union(coast, hull)
sameAsUnion <- coast + hull
plot(un)
plot(sameAsUnion)

inter <- intersect(coast, hull)
sameAsIntersect <- coast * hull
plot(inter)
plot(sameAsIntersect)

er <- erase(coast, hull)
sameAsErase <- coast - hull
plot(er)
plot(sameAsErase)

xr <- xor(coast, hull)
sameAsXor <- coast / hull
plot(xr)
plot(sameAsXor)

# Vector area and length:
```

```
expanse(coast, unit = "km") # polygons areas
expanse(rivers, unit = "km") # river lengths

### Fill holes

# First, we will make some holes by creating buffers around points.
buffs <- buffer(dypsis, 500)

holes <- coast - buffs
plot(holes)

filled <- fillHoles(holes, fail = FALSE)

}
```

---

nlevels,GRaster-method

Number of categories in a categorical raster

Description

This function reports the number of categories (levels) in a categorical GRaster.

Usage

```
## S4 method for signature 'GRaster'
nlevels(x)
```

Arguments

x	A GRaster.

Value

A named, numeric vector of integers. The values represent the number of categories (rows) that appear in the raster's levels table.

See Also

levels(), terra::levels(), droplevels(), vignette("GRasters", package = "fasterRaster")

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data: Land cover raster
```

```
madCover <- fastData("madCover")

# Convert categorical SpatRaster to categorical GRaster:
cover <- fast(madCover)

### Properties of categorical rasters

cover # note categories
is.factor(cover) # Is the raster categorical?
nlevels(cover) # number of levels
levels(cover) # just the value and active column
cats(cover) # all columns
minmax(cover) # min/max values
minmax(cover, levels = TRUE) # min/max categories
catNames(cover) # column names of the levels table
missingCats(cover) # categories in table with no values in raster
freq(cover) # frequency of each category (number of cells)
zonalGeog(cover) # geometric statistics

### Active column

# Which column sets the category labels?
activeCat(cover)
activeCat(cover, names = TRUE)

activeCats(c(cover, cover))

# Choose a different column for category labels:
levels(cover)
activeCat(cover) <- 2
levels(cover)

### Managing levels tables

# Remove unused levels:
nlevels(cover)
cover <- droplevels(cover)
nlevels(cover)

# Re-assign levels:
value <- c(20, 30, 40, 50, 120, 130, 140, 170)
label <- c("Cropland", "Cropland", "Forest", "Forest",
 "Grassland", "Shrubland", "Herbaceous", "Flooded")

newCats <- data.frame(value = value, label = label)

cover <- categories(cover, layer = 1, value = newCats)
cats(cover)

# This is the same as:
levels(cover) <- newCats
cats(cover)
```

```
# Are there any values not assigned a category?
missingCats(cover)

# Let's assign a category for value 210 (water):
water <- data.frame(value = 210, label = "Water")
addCats(cover) <- water
levels(cover)

# Add more information to the levels table using merge():
landType <- data.frame(
     Value = c(20, 30, 40, 50, 120),
     Type = c("Irrigated", "Rainfed", "Broadleaf evergreen",
     "Broadleaf deciduous", "Mosaic with forest")
)
cats(cover)
cover <- addCats(cover, landType, merge = TRUE)
cats(cover)

### Logical operations on categorical rasters

cover < "Forest" # 1 for cells with a value < 40, 0 otherwise
cover <= "Forest" # 1 for cells with a value < 120, 0 otherwise
cover == "Forest" # 1 for cells with value of 40-120, 0 otherwise
cover != "Forest" # 1 for cells with value that is not 40-120, 0 otherwise
cover > "Forest" # 1 for cells with a value > 120, 0 otherwise
cover >= "Forest" # 1 for cells with a value >= 120, 0 otherwise

cover %in% c("Cropland", "Forest") # 1 for cropland/forest cells, 0 otherwise

### Combine categories from different rasters

# For the example, will create a second categorical raster fromm elevation.

# Divide elevation raster into "low/medium/high" levels:
madElev <- fastData("madElev")
elev <- fast(madElev)
elev <- project(elev, cover, method = "near") # convert to same CRS
fun <- "= if(madElev < 100, 0, if(madElev < 400, 1, 2))"
elevCat <- app(elev, fun)

levs <- data.frame(
     value = c(0, 1, 2),
     elevation = c("low", "medium", "high")
)
levels(elevCat) <- list(levs)

# Combine levels:
combined <- concats(cover, elevCat)
combined
levels(combined)

# Combine levels, treating value/NA combinations as new categories:
combinedNA <- concats(cover, elevCat, na.rm = FALSE)
```

```
combinedNA
levels(combinedNA)

}
```

`pairs,GRaster-method` *Scatterplot of values in each GRaster layer against the others*

Description

`pairs()` generates a scatterplot between values of cells in each layer of a `GRaster` against all the other layers.

Usage

```
## S4 method for signature 'GRaster'
pairs(x, n = NULL, ...)
```

Arguments

`x`	A `GRaster` with two or more layers.
`n`	A numeric integer, integer, or `NULL` (default): Number of cells to sample. If `NULL`, 50% of the total number of cells will be used.
`...`	Arguments to send to `graphics::pairs()` (which typically sends them to `graphics::plot()`). Special arguments for affecting certain panels include: • Correlation panels – `cor.cex`: Size of the values of the correlation coefficients. • Histogram panels – `hist.col`: Histogram color. • Dot-plot panels – `colramp`: Function taking an integer `n` as input and returning `n` names of colors. The default is: `colorRampPalette(c("white", blues9))`.

Value

Nothing (creates a plot).

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")
madForest2000 <- fastData("madForest2000")
```

```
# Convert SpatRasters to GRasters
elev <- fast(madElev)
forest <- fast(madForest2000)

### GRaster properties

# plotting
plot(elev)

# dimensions
dim(elev) # rows, columns, depths, layers
nrow(elev) # rows
ncol(elev) # columns
ndepth(elev) # depths
nlyr(elev) # layers

res(elev) # resolution (2D)
res3d(elev) # resolution (3D)
zres(elev) # vertical resolution
xres(elev) # vertical resolution
yres(elev) # vertical resolution
zres(elev) # vertical resolution (NA because this is a 2D GRaster)

# cell counts
ncell(elev) # cells
ncell3d(elev) # cells (3D rasters only)

# number of NA and non-NA cells
nacell(elev)
nonnacell(elev)

# topology
topology(elev) # number of dimensions
is.2d(elev) # is it 2-dimensional?
is.3d(elev) # is it 3-dimensional?

minmax(elev) # min/max values

# "names" of the object
names(elev)

# coordinate reference system
crs(elev)
st_crs(elev)
coordRef(elev)

# extent (bounding box)
ext(elev)

# vertical extent (not defined for this raster)
zext(elev)

# data type
```

```
datatype(elev) # fasterRaster type
datatype(elev, "GRASS") # GRASS type
datatype(elev, "terra") # terra type
datatype(elev, "GDAL") # GDAL type

is.integer(elev)
is.float(elev)
is.double(elev)
is.factor(elev)

# convert data type
as.int(elev) # integer; note that "elev" is already of type "integer"
as.float(elev) # floating-precision
as.doub(elev) # double-precision

# assigning
pie <- elev
pie[] <- pi # assign all cells to the value of pi
pie

# concatenating multiple GRasters
rasts <- c(elev, forest)
rasts

# subsetting
rasts[[1]]
rasts[["madForest2000"]]

# replacing
rasts[[2]] <- 2 * forest
rasts

# adding layers
rasts[[3]] <- elev > 500 # add a layer
rasts <- c(rasts, sqrt(elev)) # add another
add(rasts) <- ln(elev)
rasts

# names
names(rasts)
names(rasts) <- c("elev_meters", "2_x_forest", "high_elevation", "sqrt_elev", "ln_elev")
rasts

# remove a layer
rasts[["2_x_forest"]] <- NULL
rasts

# number of layers
nlyr(rasts)

# correlation and covariance matrices
madLANDSAT <- fastData("madLANDSAT")
landsat <- fast(madLANDSAT) # projects matrix
```

```
layerCor(landsat) # correlation
layerCor(landsat, fun = 'cov') # covariance

}
```

---

pcs *Retrieve a principal components model from a PCA GRaster*

Description

Retrieve a principal components model from a PCA GRaster

Usage

```
pcs(x)
```

Arguments

x	A GRaster created by princomp()

Value

An object of class prcomp.

See Also

princomp(), terra::princomp(), module i.pca in **GRASS**

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Climate raster:
madChelsa <- fastData("madChelsa")

# Convert a SpatRaster to a GRaster:
chelsa <- fast(madChelsa)

# Generate raster with layers representing principal component predictions:
pcRast <- princomp(chelsa, scale = TRUE)
plot(pcRast)

# Get information on the PCA:
prinComp <- pcs(pcRast)

prinComp
summary(prinComp)
```

```
plot(prinComp)

}
```

`plot,GRaster,missing-method`
Display a raster or vector

Description

`plot()` displays a `GRaster` or `GVector`.

This function is essentially a hack, as it it not possible to dependably call the appropriate **GRASS** modules and display a raster or vector without potential confusion on the user side. Instead, this function 1) simplifies the focal `GRaster` or `GVector` to make it smaller when saved to disk; 2) writes the object to disk; 3) (internally) creates a `SpatRaster` or `SpatVector` object; then 4) plots the object using `terra::plot()`. Thus, if you are interested in making maps, it will always be faster to make them directly with **terra** or **sf**.

Usage

```
## S4 method for signature 'GRaster,missing'
plot(x, y, simplify = TRUE, ...)

## S4 method for signature 'GVector,missing'
plot(x, y, maxGeoms = 10000, ...)
```

Arguments

`x`	A `GRaster` or `GVector`.
`y`	Missing–leave as empty.
`simplify`	Logical: If `TRUE` (default) and the raster has an x- and/or y-resolution that is greater than the screen resolution, then `aggregate()` will be applied to the raster before it is saved to reduce the time it takes to save the raster.
`...`	Other arguments to send to `terra::plot()`.
`maxGeoms`	Positive integer (vectors only): Maximum number of features before vector simplification is applied before saving to disk then creating a `SpatVector` for plotting. The default is 10000.

Value

Nothing (displays a raster or vector).

See Also

`terra::plot()`

Examples

```
if (grassStarted()) {

# Example data
madElev <- fastData("madElev") # elevation raster
madLANDSAT <- fastData("madLANDSAT") # multi-layer raster
madRivers <- fastData("madRivers") # lines vector

# Convert SpatRaster to GRaster and SpatVector to GVector
elev <- fast(madElev)
rivers <- fast(madRivers)
landsat <- fast(madLANDSAT)

# Plot:
plot(elev)
plot(rivers, add = TRUE)

# Histograms:
hist(elev)
hist(landsat)

# Plot surface reflectance in RGB:
plotRGB(landsat, 3, 2, 1) # "natural" color
plotRGB(landsat, 4, 1, 2, stretch = "lin") # emphasize near-infrared (vegetation)

# Make composite map from RGB layers and plot in grayscale:
comp <- compositeRGB(r = landsat[[3]], g = landsat[[2]], b = landsat[[1]])
grays <- paste0("gray", 0:100)
plot(comp, col = grays)

}
```

---

plotRGB,GRaster-method

Create red-green-blue plot from a raster with RGB layers

Description

This function takes as its main argument a `GRaster` with at least three layers typically representing red, green, and blue components (plus possibly an "alpha", or transparency layer). As with `plot()`, this function is somewhat of a hack in that it downsamples the layers to a coarser resolution using `aggregate()`, saves the raster to disk, then uses `terra::plotRGB()` to do the actual plotting.

Usage

```
## S4 method for signature 'GRaster'
plotRGB(x, r = 1, g = 2, b = 3, a = NULL, simplify = TRUE, ...)
```

Arguments

x	A GRaster. Values must be in the range from 0 to 255.
r, g, b	Either a numeric integer or the names() of layers representing red, green, and blue components.
a	Either NULL (default), or a numeric integer or the names() of a layer representing transparency.
simplify	Logical: If TRUE (default), then downsample the GRaster before plotting. This can save time for very dense rasters.
...	Arguments to pass to terra::plotRGB().

Value

Nothing (makes a plot).

See Also

terra::plotRGB(), plot(), compositeRGB()

Examples

```
if (grassStarted()) {

# Example data
madElev <- fastData("madElev") # elevation raster
madLANDSAT <- fastData("madLANDSAT") # multi-layer raster
madRivers <- fastData("madRivers") # lines vector

# Convert SpatRaster to GRaster and SpatVector to GVector
elev <- fast(madElev)
rivers <- fast(madRivers)
landsat <- fast(madLANDSAT)

# Plot:
plot(elev)
plot(rivers, add = TRUE)

# Histograms:
hist(elev)
hist(landsat)

# Plot surface reflectance in RGB:
plotRGB(landsat, 3, 2, 1) # "natural" color
plotRGB(landsat, 4, 1, 2, stretch = "lin") # emphasize near-infrared (vegetation)

# Make composite map from RGB layers and plot in grayscale:
comp <- compositeRGB(r = landsat[[3]], g = landsat[[2]], b = landsat[[1]])
grays <- paste0("gray", 0:100)
plot(comp, col = grays)

}
```

predict,GRaster-method

Make predictions from a linear or generalized linear model to a GRaster

Description

This version of the `predict()` function make predictions to a set of `GRaster`s from a model object.

The model must be either a linear model, which is of class `lm` and typically created using the `stats::lm()` function or a generalized linear model (GLM), which is class `glm` and typically created using `stats::glm()`. Other packages can also create `lm` or `glm` objects, but they may not work in this function. For example, generalized additive models, which can be created using the `gam()` function in the **mgcv** package, inherit the `glm` class, but cannot be used in this function. However, `glm` objects created with the **speedglm** package should work with this function.

This `predict()` function can handle:

- Linear predictors and intercepts like `1 + x`;
- Quadratic terms like `x^2` (or, in **R** formula notation, `I(x^2)`);
- Two-way interaction terms between scalars like `x1:x2` and `x1 * x2`;
- Categorical predictors (i.e., categorical `GRaster`s; see `vignette("GRasters", package = "fasterRaster")`);
- Two-way interactions between a categorical predictor and a scalar predictor; and
- Two-way interactions between categorical predictors.

Usage

```
## S4 method for signature 'GRaster'
predict(object, model, type = "response")
```

Arguments

`object`	A `GRaster` with one or more layers.
`model`	An `lm` or `glm` model object.
`type`	Character: Type of prediction to make. This can be either `link` (default; predictions are made on the scale of the link function) or `response` (predictions are made on the scale of the response variable). This function can only make predictions on the scale of the response for the identity, logit, log, or cloglog (complementary log-log) link functions.

Value

A `GRaster`.

See Also

`terra::predict()`; `stats::predict()`

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

### This example creates a simple model of Dypsis distribution using
# elevation, distance to forest, land cover class, and nearness to rivers.

# Elevation raster, forest cover in year 2000, land cover class, and
# points where Dypsis plants have been collected
madElev <- fastData("madElev")
madForest2000 <- fastData("madForest2000")
madCover <- fastData("madCover")
madRivers <- fastData("madRivers")
madDypsis <- fastData("madDypsis")

# Convert SpatRasters to GRasters and sf vectors to GVectors:
elev <- fast(madElev)
forest <- fast(madForest2000)
cover <- fast(madCover)
rivers <- fast(madRivers)
dypsis <- fast(madDypsis)

# Distance to forest
distToForest <- distance(forest, unit = "m")
distToForest <- log1p(distToForest) # log(x + 1) of distance
names(distToForest) <- "distToForest"

# "Stack" elevation and forest cover
continuous <- c(elev, distToForest)

# Scale continuous predictors to mean of 0 and sd of 1
continuousScaled <- scale(continuous)
names(continuousScaled) <- c("elevation", "distToForest")

# Project land cover raster
coverProj <- project(cover, continuousScaled)

# Near a river?
riverBuffer <- buffer(rivers, 5000)
nearRiver <- rasterize(riverBuffer, elev, background = 0)
names(nearRiver) <- "nearRiver"
levels(nearRiver) <- data.frame(value = 0:1, label = c("far", "near"))

# Combine continuous/categorical data
covariateRasters <- c(continuousScaled, coverProj, nearRiver)
plot(covariateRasters)

# Extract environmental values at Dypsis locations:
presEnv <- extract(covariateRasters, dypsis, cats = TRUE)
```

```
presEnv$presBg <- 1
head(presEnv)

# Extract elevation and forest cover at 2000 background sites:
bgEnv <- spatSample(covariateRasters, size = 3000, values = TRUE, cats = TRUE)
bgEnv <- bgEnv[stats::complete.cases(bgEnv), ]
bgEnv <- bgEnv[1:2000, ]
bgEnv$presBg <- 0
head(bgEnv)

# Combine presence and background data:
env <- rbind(presEnv, bgEnv)

# Calibrate model:
form <- presBg ~ elevation + distToForest +
  I(distToForest^2) + elevation * distToForest +
  madCover + nearRiver

model <- stats::glm(form, data = env, family = stats::binomial)
summary(model)

# Make predictions and map:
prediction <- predict(covariateRasters, model, type = "response")
prediction

# Not a great model!
plot(prediction, main = "Predicted")
plot(dypsis, pch = 1, add = TRUE)

}
```

`princomp,GRaster-method`

Apply a principal component analysis (PCA) to layers of a GRaster

---

Description

This function applies a principal component analysis to layers of a `GRaster`.

Usage

```
## S4 method for signature 'GRaster'
princomp(x, scale = TRUE, scores = FALSE)
```

Arguments

`x`	A `GRaster` with two or more layers.

scale — Logical: If TRUE (default), input layers will be rescaled by dividing each layer by its overall population standard deviation. Rasters will always be centered (have their mean subtracted from values). Centering and scaling is recommended when rasters values are in different units.

scores — Logical: If TRUE, the prcomp object will have the scores attached to it. This can greatly increase the size of the object in memory if the input raster has many cells. It will also take more time. If FALSE (default), then skip returning scores.

Value

A multi-layer GRaster with one layer per principal component axis. The pcs() function can be used on the output raster to retrieve a prcomp object from the raster, which includes rotations (loadings) and proportions of variance explained.

See Also

terra::princomp(), terra::prcomp()

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Climate raster:
madChelsa <- fastData("madChelsa")

# Convert a SpatRaster to a GRaster:
chelsa <- fast(madChelsa)

# Generate raster with layers representing principal component predictions:
pcRast <- princomp(chelsa, scale = TRUE)
plot(pcRast)

# Get information on the PCA:
prinComp <- pcs(pcRast)

prinComp
summary(prinComp)
plot(prinComp)

}
```

---

project,GRaster-method

Change the coordinate reference system of a GRaster or GVector

---

Description

`project()` changes the coordinate reference system (CRS) of a `GRaster` or `GVector`. It has three use cases:

- `x` is a `GRaster` and `y` is a `GRaster`: `x` will be projected to the CRS of `y` and resampled to have the same resolution as `y`. If argument `align` is `FALSE`, then it will also be cropped to the extent of `y`.
- `x` is a `GRaster` and `y` is a `GVector` or a CRS string (typically in Well-Known Text format): `x` will be projected to the CRS specified by `y` and resampled but not cropped.
- `x` is a `GVector` and `y` is a `GRaster`, `GVector`, or CRS string: The vector will be projected to the CRS of `y`.

Usage

```
## S4 method for signature 'GRaster'
project(
  x,
  y,
  align = FALSE,
  method = NULL,
  fallback = TRUE,
  res = "fallback",
  wrap = FALSE,
  verbose = FALSE
)

## S4 method for signature 'GVector'
project(x, y, wrap = FALSE)
```

Arguments

`x`	A `GRaster` or `GVector` to be projected.
`y`	A character or `GLocation` object (i.e., typically a `GRaster` or `GVector`): Used to set the focal `GRaster` or `GVector`'s new CRS (and resolution and possibly extent, for `GRasters`).
`align`	Logical: If `FALSE` (default), and `x` and `y` are `GRasters`, then the extent of `x` will be cropped to the extent of `y`. If `TRUE`, no cropping is performed.
`method`	Character or `NULL` (for `GRasters` only): Method to use to conduct the transformation (rasters only). Partial matching is used. • `NULL` (default): Automatically choose based on raster properties (`near` for categorical data, `bilinear` for continuous data). • `"near"`: Nearest neighbor. Best for categorical data, and often a poor choice for continuous data. If `datatype()` is `integer`, this method will be used by default. • `"bilinear"`: Bilinear interpolation (default for non-categorical data; uses weighted values from 4 cells).

- "bicubic": Bicubic interpolation (uses weighted values from 16 cells).
- "lanczos": Lanczos interpolation (uses weighted values from 25 cells).

Note #1: If x and y are GRasters, and res = "terra", then the same method is used to resample x to the resolution of y before projecting x.

Note #2: Methods that use multiple cells will cause the focal cell to become NA if there is at least one cell with an NA in the cells it draws from. These NA cells can often be filled using the fallback argument.

fallback — Logical (for projecting GRasters only): If TRUE (default), then use "lower" resampling methods to fill in NA cells when a "higher" resampling method is used. For example, if method = "bicubic", NA cells will be filled in using the bilinear method, except when that results in NAs, in which case the near method will be used. Fallback causes fewer cells to revert to NA values, so may be better at capturing complex "edges" (e.g., coastlines). Fallback does increase processing time because each "lower" method must be applied, then results merged. Fallback is not used if method = "near".

res — Character (for projecting GRasters only): Method used to set the resolution of a GRaster in the new CRS. This can be one of three options. Partial matching is used and case ignored:

- "terra": This method creates an output raster that is as close as possible in values and resolution to the one that terra::project() would create. However, for large rasters (i.e., many cells), this can fail because terra::project() encounters memory limits (it is used internally to create a template). This method resamples the focal raster in its starting CRS, then projects it to the destination CRS.
- "template": This method can only be used if y is a GRaster. The output will have the same resolution as y and possibly the same extent (depending on the value of align). However, unlike the "terra" method, cell values will not necessarily be as close as possible to what terra::project() would generate (unless method = "near"). Unlike the "terra" method, this method does not resample the focal raster in its starting CRS before projecting. For large rasters it will be faster than the "terra" method (especially if "method = "near"), and it should be less likely to fail because of memory limits.
- Two numeric values: Values for the new resolution (x- and y-dimensions).
- "center": This method locates the centroid of the raster to be projected (in the same CRS as the original raster). It then creates four points north, south, east, and west of the centroid, each spaced one cell's width from the centroid. This set of points is then projected to the new CRS. The new cell size in the x-dimension will be the average of the distance between the east and west points from the centroid, and in the y-dimension the average from the centroid to the north and south points.
- "fallback" (default): This applies the terra method first, but if that fails, then tries template, then center. This process can take a long time for large rasters.

wrap — Logical:

- GRasters: When projecting rasters that "wrap around" (i.e., whole-world rasters or rasters that have edges that actually circle around to meet on the globe), wrap should be TRUE to avoid removing rows and columns from the "edge" of the map. The default is FALSE.
- GVectors: When projecting vectors that span the international date line at 180E/W, wrap should be TRUE to avoid an issue where the coordinates are incorrectly mapped to the range -180 to 180.

verbose — Logical (for projecting GRasters only): If TRUE, display progress. Default is FALSE.

Details

When projecting a raster, the "fallback" methods in **GRASS** module r.import are actually used, even though the method argument takes the strings specifying non-fallback methods. See the manual page for the r.import **GRASS** module.

Value

A GRaster or GVector.

See Also

terra::project(), sf::st_transform(), **GRASS** manual pages for modules r.proj and v.proj (see grassHelp("r.proj") and grassHelp("v.proj"))

Examples

```
if (grassStarted()) {

### Setup for all examples

library(sf)
library(terra)

# Climate raster, elevation raster, rivers vector
madElev <- fastData("madElev")
madRivers <- fastData("madRivers")
madChelsa <- fastData("madChelsa")

# Convert objects into fasterRaster formats
chelsa <- fast(madChelsa)
elev <- fast(madElev)
rivers <- fast(madRivers)

### Project raster without resampling
elevWGS84 <- project(elev, crs(chelsa))
elevWGS84

### Project raster and resample to resolution of another raster
elevWGS84Resamp <- project(elev, chelsa)
elevWGS84Resamp
```

```
res(elevWGS84)
res(elevWGS84Resamp)
res(chelsa)

### Project vector
riversWGS84 <- project(rivers, chelsa)
riversWGS84
cat(crs(rivers)) # using "cat()" to make it look nice
cat(crs(riversWGS84))

}
```

`rast,GRaster-method` *Convert a GRaster to a SpatRaster*

Description

The **fasterRaster** version of the `rast()` function converts a `GRaster` to a `SpatRaster` (from the **terra** package).

Usage

```
## S4 method for signature 'GRaster'
rast(x, mm = FALSE, ...)
```

Arguments

`x`	A `GRaster`.
`mm`	Logical: If `TRUE`, call `terra::setMinMax()` on the raster to ensure it has metadata on the minimum and maximum values. For large rasters, this can take a long time, so the default value of `mm` is `FALSE`.
`...`	Additional arguments to send to `writeRaster()`. These are typically unneeded, though `bigTiff` may be of use if the raster is large, and supplying `datatype` can speed conversion of large rasters. See `writeRaster()`.

Value

A `SpatRaster` (**terra** package).

See Also

`terra::rast()`

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")
madForest2000 <- fastData("madForest2000")

# Convert SpatRasters to GRasters
elev <- fast(madElev)
forest <- fast(madForest2000)

### GRaster properties

# plotting
plot(elev)

# dimensions
dim(elev) # rows, columns, depths, layers
nrow(elev) # rows
ncol(elev) # columns
ndepth(elev) # depths
nlyr(elev) # layers

res(elev) # resolution (2D)
res3d(elev) # resolution (3D)
zres(elev) # vertical resolution
xres(elev) # vertical resolution
yres(elev) # vertical resolution
zres(elev) # vertical resolution (NA because this is a 2D GRaster)

# cell counts
ncell(elev) # cells
ncell3d(elev) # cells (3D rasters only)

# number of NA and non-NA cells
nacell(elev)
nonnacell(elev)

# topology
topology(elev) # number of dimensions
is.2d(elev) # is it 2-dimensional?
is.3d(elev) # is it 3-dimensional?

minmax(elev) # min/max values

# "names" of the object
names(elev)

# coordinate reference system
```

```
crs(elev)
st_crs(elev)
coordRef(elev)

# extent (bounding box)
ext(elev)

# vertical extent (not defined for this raster)
zext(elev)

# data type
datatype(elev) # fasterRaster type
datatype(elev, "GRASS") # GRASS type
datatype(elev, "terra") # terra type
datatype(elev, "GDAL") # GDAL type

is.integer(elev)
is.float(elev)
is.double(elev)
is.factor(elev)

# convert data type
as.int(elev) # integer; note that "elev" is already of type "integer"
as.float(elev) # floating-precision
as.doub(elev) # double-precision

# assigning
pie <- elev
pie[] <- pi # assign all cells to the value of pi
pie

# concatenating multiple GRasters
rasts <- c(elev, forest)
rasts

# subsetting
rasts[[1]]
rasts[["madForest2000"]]

# replacing
rasts[[2]] <- 2 * forest
rasts

# adding layers
rasts[[3]] <- elev > 500 # add a layer
rasts <- c(rasts, sqrt(elev)) # add another
add(rasts) <- ln(elev)
rasts

# names
names(rasts)
names(rasts) <- c("elev_meters", "2_x_forest", "high_elevation", "sqrt_elev", "ln_elev")
rasts
```

```
# remove a layer
rasts[["2_x_forest"]] <- NULL
rasts

# number of layers
nlyr(rasts)

# correlation and covariance matrices
madLANDSAT <- fastData("madLANDSAT")
landsat <- fast(madLANDSAT) # projects matrix
layerCor(landsat) # correlation
layerCor(landsat, fun = 'cov') # covariance

}
```

`rasterize,GVector,GRaster-method`

Convert a GVector to a GRaster

Description

The `rasterize()` function converts a `GVector` into a `GRaster`.

Usage

```
## S4 method for signature 'GVector,GRaster'
rasterize(x, y, field = "", background = NA, by = NULL, verbose = TRUE)
```

Arguments

`x`	A `GVector`.
`y`	A `GRaster`: The new raster will have the same extent and resolution as this raster.
`field`	Character: Name of a column in the data table of `y` to "burn" into the raster. If not `""` (default), then the output will be a categorical `GRraster`. If `field` is `""`, then all geometries will be "burned" to the raster and have the same value.
`background`	Numeric or `NA` (default): Value to put in cells that are not covered by the `GVector`. Note that if this is not `NA` and not an integer, then the output cannot be a categorical raster (i.e., there will be no "levels" table associated with it).
`by`	Either `NULL` (default) or character: If this is not `NULL`, then the `GVector` will be subset by the values in the field named by `by`. The output will be a multi-layer raster, with one layer per unique value in `by`.
`verbose`	Logical: If `by` is not `NULL`, display progress.

Value

A `GRaster`.

See Also

terra::rasterize(), **GRASS** module v.to.rast (see grassHelp("v.to.rast"))

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Elevation raster, outline of a part of Madagascar, and rivers vector:
madElev <- fastData("madElev") # raster
madDypsis <- fastData("madDypsis") # points vector
madRivers <- fastData("madRivers") # lines vector
madCoast4 <- fastData("madCoast4") # polygons vector

# Convert to GRaster and GVectors:
elev <- fast(madElev)
dypsis <- fast(madDypsis)
coast4 <- fast(madCoast4)
rivers <- fast(madRivers)

# Convert points, line, and polygons vectors to rasters:
points <- rasterize(dypsis, elev)
plot(points)

lines <- rasterize(rivers, elev)
plot(lines)

polys <- rasterize(coast4, elev)
plot(polys)

communes <- rasterize(coast4, elev, field = "NAME_4")
plot(communes)

# Change background value:
polysNeg1 <- rasterize(coast4, elev, background = -1)
plot(polysNeg1)

# Make one layer per river:
byRiver <- rasterize(rivers, elev, field = "NAM", by = "NAM")
plot(byRiver)

}
```

---

rbind,GVector-method *Combine one or more GVectors*

Description

rbind() combines two or more GVectors of the same type (points, lines, or polygons) and same coordinate reference system. You can speed operations by putting the vector that is largest in memory first in rbind(...). If the GVectors have data tables, these will also be combined using rbind() if their column names and data types match.

Usage

```
## S4 method for signature 'GVector'
rbind(..., deparse.level = 1)
```

Arguments

...	One or more GVectors.
deparse.level	See rbind().

Value

A GVector.

See Also

colbind(), addTable<-, dropTable()

Examples

```
if (grassStarted()) {

# Setup
library(sf)

# Rivers vector
madRivers <- fastData("madRivers")

# Convert sf to a GVector
rivers <- fast(madRivers)

# Convert GVector to data.frame or data.table
as.data.frame(rivers)
as.data.table(rivers)

# Subset rivers vector
rivers1 <- rivers[1:2]
rivers2 <- rivers[10:11]

# Concatenate rivers
riversCombo <- rbind(rivers1, rivers2)
riversCombo

# Add columns
newCol <- data.frame(new = 1:11)
```

```
riversCol <- colbind(rivers, newCol)
riversCol

# Remove table
riversCopy <- rivers
riversCopy # has data table
riversCopy <- dropTable(riversCopy)
riversCopy # no data table

# Add a new table
newTable <- data.frame(num = 1:11, letters = letters[1:11])
addTable(riversCopy) <- newTable
riversCopy

}
```

`regress,GRaster,missing-method`

Regression intercept, slope, r2, and t-value across each set of cells

---

Description

This function performs a regression on each set of cells in a multi-layered `GRaster`. The output is a `GRaster` with the intercept, slope, r^2 value, and Student's t value. The regression formula is as `y ~ 1 + x`, where `x` is the layer number of each layer (e.g., 1 for the first or top layer in the input `GRaster`, 2 for the second or second-to-top layer, etc.). Note that this is restricted version of the functionality in `terra::regress()`.

Usage

```
## S4 method for signature 'GRaster,missing'
regress(y, x, na.rm = FALSE)
```

Arguments

`y`	A multi-layer `GRaster`.
`x`	Ignored.
`na.rm`	Logical: If `FALSE`, any series of cells with `NA` in at least one cell results in an `NA` in the output.

Value

A multi-layer `GRaster`.

See Also

`terra::regress()`

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Example data
madChelsa <- fastData("madChelsa")

# Convert a SpatRaster to a GRaster
chelsa <- fast(madChelsa)
chelsa # 4 layers

# Central tendency
mean(chelsa)
mmode(chelsa)
median(chelsa)

# Statistics
nunique(chelsa)
sum(chelsa)
count(chelsa)
min(chelsa)
max(chelsa)
range(chelsa)
skewness(chelsa)
kurtosis(chelsa)

stdev(chelsa)
stdev(chelsa, pop = FALSE)
var(chelsa)
varpop(chelsa)

# Which layers have maximum/minimum?
which.min(chelsa)
which.max(chelsa)

# Regression

# Note the intercept is different for fasterRaster::regress().
regress(chelsa)
regress(madChelsa, 1:nlyr(madChelsa))

# Note: To get quantiles for each layer, use global().
quantile(chelsa, 0.1)

# NAs
madForest2000 <- fastData("madForest2000")
forest2000 <- fast(madForest2000)
forest2000 <- project(forest2000, chelsa, method = "near")
```

```
chelsaForest <- c(chelsa, forest2000)

nas <- anyNA(chelsaForest)
plot(nas)

allNas <- allNA(chelsaForest)
plot(allNas)

}
```

`reorient,GRaster-method`
Convert degrees between 'north-orientation' and 'east orientation'

Description

This function converts facing between "north orientation" and "east orientation".

In "north orientation" systems, a 0-degree facing is north, and the angle of facing proceeds clockwise. For example, a 90 degree facing faces east, 180 south, and 270 west. In "east orientation", a 0-degree facing is east, and the facing angle proceeds counter-clockwise. For example, 90 is north, 180 is west, and 270 south.

Usage

```
## S4 method for signature 'GRaster'
reorient(x, units = "degrees")

## S4 method for signature 'numeric'
reorient(x, units = "degrees")
```

Arguments

`x`	A numeric vector or a `GRaster` with cell values equal to facing (in degrees).
`units`	Character: "Units" of values in `x`: either `"degrees"` for degrees (default) or `"radians"`. Partial matching is used.

Value

A `GRaster` or numeric vector. Values will be in the range between 0 and 360 and represents facing in the system "opposing" the input's system. For example, if the input is north orientation, the output will be east orientation. If the input is in east orientation, the output will be in north orientation.

Examples

```
### Re-orient numeric values:
facings <- c(0, 90, 180, 270, 360)
reorient(facings)

# Re-reorienting returns the same values:
reorient(reorient(facings))

if (grassStarted()) {

### Re-orient a GRaster:

# Setup
library(terra)
madElev <- fastData("madElev")
elev <- fast(madElev)

# Calculate aspect in degrees, using north orientation:
aspectNorth <- terrain(elev, "aspect")

# Re-orient to east-facing:
aspectEast <- reorient(aspectNorth)

# Re-reorienting is the same, to within rounding error:
aspectNorth - reorient(reorient(aspectNorth))

# Plot:
aspects <- c(aspectNorth, aspectEast)
names(aspects) <- c("north_orientation", "east_orientation")
plot(aspects)


}
```

---

replaceNAs,data.frame-method

Replace NAs in a data.table or data.frame column, or in a vector

---

Description

This function replaces NAs in one or more `data.table`, `data.frame`, or `matrix` columns, or in vectors, with a user-defined value.

Usage

```
## S4 method for signature 'data.frame'
replaceNAs(x, replace, cols = NULL)

## S4 method for signature 'matrix'
```

```
replaceNAs(x, replace, cols = NULL)

## S4 method for signature 'data.table'
replaceNAs(x, replace, cols = NULL)

## S4 method for signature 'numeric'
replaceNAs(x, replace)

## S4 method for signature 'integer'
replaceNAs(x, replace)

## S4 method for signature 'logical'
replaceNAs(x, replace)

## S4 method for signature 'character'
replaceNAs(x, replace)
```

Arguments

x	A data.table or data.frame or matrix, or a vector of numeric, integer, logical, or character values.
replace	A value of any atomic class (numeric, integer, character, Date, etc.): Value to to replace NAs.
cols	NULL, character, numeric, integer, or logical vector: Indicates columns for which to replace NAs. If NULL, then all columns will have NAs replaced. If a character, this is the column name(s). If numeric or integer, this is the columns' indices. If logical, columns with TRUE have NAs replaced. If a logical vector has fewer than the total number of columns, it will be recycled.

Value

A data.table, data.frame, matrix, or vector.

Examples

```
library(data.table)

dt <- data.table(
   x = 1:10,
   y = letters[1:10],
   z = rnorm(10)
)

# make some values NA
dt[x == 4 | x == 8, y := NA_character_]
dt

# Replace NAs:
replaceNAs(dt, replace = -99, cols = "y")
dt
```

```
# Drop rows:
dropped <- dropRows(dt, 8:10)
dropped

# NB May not print... in that case, use:
print(dropped)

# We can also use replaceNAs() on vectors:
y <- 1:10
y[c(2, 10)] <- NA
replaceNAs(y, -99)

# Same as:
y <- 1:10
y[c(2, 10)] <- NA
y[is.na(y)] <- -99
```

`res,missing-method` *Spatial resolution*

Description

Spatial resolution of a `GRaster`:

- `res()`: 2-dimensional resolution (x and y).
- `res3d()`: 3-dimensional resolution (z, y, and z).
- `xres()`, `yres()`, and `zres()`: East-west resolution, north-south resolution, and top-bottom resolution.

Usage

```
## S4 method for signature 'missing'
res(x)

## S4 method for signature 'GRegion'
res(x)

## S4 method for signature 'missing'
xres(x)

## S4 method for signature 'GRegion'
xres(x)
```

```
## S4 method for signature 'missing'
yres(x)

## S4 method for signature 'GRegion'
yres(x)

## S4 method for signature 'missing'
zres(x)

## S4 method for signature 'GRegion'
zres(x)

## S4 method for signature 'missing'
res3d(x)

## S4 method for signature 'GRegion'
res3d(x)
```

Arguments

x	A GRaster, GRegion, or missing. If missing, the resolution of the currently active "region" is returned (see vignette("regions", package = "fasterRaster")).

Value

A numeric vector. For both res() and res3d(), the first value is the length of cells in the x-direction and the second the length of cells in the y-direction. For res3d() the third value is height of a voxel (the z-direction). xres(), yres(), and zres() each return a single value.

See Also

terra::res()

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")
madForest2000 <- fastData("madForest2000")

# Convert SpatRasters to GRasters
elev <- fast(madElev)
forest <- fast(madForest2000)

### GRaster properties

# plotting
```

```
plot(elev)

# dimensions
dim(elev) # rows, columns, depths, layers
nrow(elev) # rows
ncol(elev) # columns
ndepth(elev) # depths
nlyr(elev) # layers

res(elev) # resolution (2D)
res3d(elev) # resolution (3D)
zres(elev) # vertical resolution
xres(elev) # vertical resolution
yres(elev) # vertical resolution
zres(elev) # vertical resolution (NA because this is a 2D GRaster)

# cell counts
ncell(elev) # cells
ncell3d(elev) # cells (3D rasters only)

# number of NA and non-NA cells
nacell(elev)
nonnacell(elev)

# topology
topology(elev) # number of dimensions
is.2d(elev) # is it 2-dimensional?
is.3d(elev) # is it 3-dimensional?

minmax(elev) # min/max values

# "names" of the object
names(elev)

# coordinate reference system
crs(elev)
st_crs(elev)
coordRef(elev)

# extent (bounding box)
ext(elev)

# vertical extent (not defined for this raster)
zext(elev)

# data type
datatype(elev) # fasterRaster type
datatype(elev, "GRASS") # GRASS type
datatype(elev, "terra") # terra type
datatype(elev, "GDAL") # GDAL type

is.integer(elev)
is.float(elev)
```

```
is.double(elev)
is.factor(elev)

# convert data type
as.int(elev) # integer; note that "elev" is already of type "integer"
as.float(elev) # floating-precision
as.doub(elev) # double-precision

# assigning
pie <- elev
pie[] <- pi # assign all cells to the value of pi
pie

# concatenating multiple GRasters
rasts <- c(elev, forest)
rasts

# subsetting
rasts[[1]]
rasts[["madForest2000"]]

# replacing
rasts[[2]] <- 2 * forest
rasts

# adding layers
rasts[[3]] <- elev > 500 # add a layer
rasts <- c(rasts, sqrt(elev)) # add another
add(rasts) <- ln(elev)
rasts

# names
names(rasts)
names(rasts) <- c("elev_meters", "2_x_forest", "high_elevation", "sqrt_elev", "ln_elev")
rasts

# remove a layer
rasts[["2_x_forest"]] <- NULL
rasts

# number of layers
nlyr(rasts)

# correlation and covariance matrices
madLANDSAT <- fastData("madLANDSAT")
landsat <- fast(madLANDSAT) # projects matrix
layerCor(landsat) # correlation
layerCor(landsat, fun = 'cov') # covariance

}
```

resample,GRaster,GRaster-method

Change the cell size of a GRaster

Description

`resample()` changes the cell size (resolution) of a `GRaster` using either another raster as a template or a user-defined resolution. Note that the extent of the output raster may be expanded to accommodate an integer number of cells. The function is not guaranteed to recreate the same output as `terra::resample()`, even when the same resampling method is used.

Usage

```
## S4 method for signature 'GRaster,GRaster'
resample(x, y, method = NULL, fallback = TRUE)

## S4 method for signature 'GRaster,numeric'
resample(x, y, method = NULL, fallback = TRUE)
```

Arguments

`x`	The `GRaster` to resample.
`y`	Either a `GRaster` to serve as a template, or a numeric vector with two or three values. If a numeric vector, the values represent east-west and north-south resolution for 2D rasters, or east-west, north-south, and top-bottom resolution for 3D rasters.
`method`	Character or `NULL`: Method to use to assign values to cells. Partial matching is used. • `NULL` (default): Automatically choose based on raster properties (`near` for categorical or integer rasters, `bilinear` for continuous data). • `"near"`: Nearest neighbor. Best for categorical data, and often a poor choice for continuous data. If `nlevels()` is >0, this method will be used regardless of the value of `method`. If you still want to use a different method, coerce the raster to a different type using `as.int()`, `as.float()`, or `as.doub()`. • `"bilinear"`: Bilinear interpolation (default for non-categorical data; uses weighted values from 4 cells). • `"bicubic"`: Bicubic interpolation (uses weighted values from 16 cells). • `"lanczos"`: Lanczos interpolation (uses weighted values from 25 cells). Note that methods that use multiple cells will cause the focal cell to become `NA` if there is at least one cell with an `NA` in the cells it draws from. These `NA` cells can be filled using the `fallback` option.
`fallback`	Logical: If `TRUE` (default), then use "lower" methods to fill in `NA` cells when a "higher" method is used. For example, if `method = "bicubic"`, `NA` cells will be filled in using the `bilinear` method, except when that results in `NA`s, in which

case the near method will be used. Fallback causes fewer cells to revert to NA values, so can be better at resampling the edges of rasters. However, fallback does increase processing time because each "lower" method must be applied, then results merged.

Value

A GRaster.

See Also

terra::resample(), **GRASS** modules r.resample and r.resamp.interp (see grassHelp("r.resample") and grassHelp(

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Elevation raster
madElev <- fastData("madElev")
elev <- fast(madElev)

### Resample raster to 120 x 120 m
elev120 <- resample(elev, c(120, 120), method="bilinear")
elev
elev120

### Resample using another raster as a template

template <- aggregate(elev, 4)

nearest <- resample(elev, template, method = "nearest")

bilinear <- resample(elev, template, method = "bilinear")
bilinearNoFB <- resample(elev, template, method = "bilinear", fallback = FALSE)

bicubic <- resample(elev, template, method = "bicubic")
bicubicNoFB <- resample(elev, template, method = "bicubic", fallback = FALSE)

lanczos <- resample(elev, template, method = "lanczos")
lanczosNoFB <- resample(elev, template, method = "lanczos", fallback = FALSE)

# rasters resampled without fallback have fewer non-NA cells
resampled <- c(nearest, bilinear, bilinearNoFB, bicubic, bicubicNoFB, lanczos,
    lanczosNoFB)
names(resampled) <- c("nearest", "bilinear", "bilinearNoFB", "bicubic",
    "bicubicNoFB", "lanczos", "lanczosNoFB")
ones <- resampled * 0 + 1
global(ones, "sum") # number of non-NA cells
global(resampled, c("mean", "sd", "min", "max")) # other statistics
```

```
# Compare fallback to no fallback
frLanczos <- rast(lanczos)
frLanczosNoFB <- rast(lanczosNoFB)

plot(frLanczos, col = "red",
    main = "Red: Cells in fallback not non-fallback", legend = FALSE)
plot(frLanczosNoFB, add=TRUE)

# Compare fasterRaster with terra
coarserTerra <- aggregate(madElev, 4)
terraLanczos <- resample(madElev, coarserTerra, method = "lanczos")

frLanczos <- extend(frLanczos, terraLanczos)
frLanczosNoFB <- extend(frLanczosNoFB, terraLanczos)

frLanczos - terraLanczos
frLanczosNoFB - terraLanczos

plot(frLanczos - terraLanczos, main = "Difference")
plot(frLanczosNoFB - terraLanczos, main = "Difference")

plot(terraLanczos, col = "red",
    main = "Red: Cells in terra not in FR", legend = FALSE)
plot(frLanczos, add=TRUE)

plot(frLanczos, col = "red",
    main = "Red: Cells in FR not in terra", legend = FALSE)
plot(terraLanczos, add=TRUE)

}
```

`rnormRast,GRaster-method`

Create a raster with random values drawn from a normal distribution

---

Description

`rnormRast()` creates a raster with values drawn from a normal distribution.

Usage

```
## S4 method for signature 'GRaster'
rnormRast(x, n = 1, mu = 0, sigma = 1, seed = NULL)
```

Arguments

`x`	A `GRaster`: The output will have the same extent and dimensions as this raster.
`n`	An integer: Number of rasters to generate.

mu, sigma	Numeric: Mean and sample standard deviation of output. If creating more than one raster, you can provide one value per raster. If there are fewer, they will be recycled.
seed	Numeric integer or NULL: Random seed. If NULL, then a random seed is used for each raster. If provided, there should be one seed value per raster.

Value

A GRaster.

See Also

rSpatialDepRast(), fractalRast(), runifRast(), **GRASS** manual page for module r.random.surface (see grassHelp("r.random.surface"))

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Elevation raster
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster:
elev <- fast(madElev)

### Create a raster with values drawn from a uniform distribution:
unif <- runifRast(elev)
plot(unif)

### Create a raster with values drawn from a normal distribution:
norms <- rnormRast(elev, n = 2, mu = c(5, 10), sigma = c(2, 1))
plot(norms)
hist(norms, bins = 100)

# Create a raster with random, seemingly normally-distributed values:
rand <- rSpatialDepRast(elev, dist = 1000)
plot(rand)

# Values appear normal on first inspection:
hist(rand)

# ... but actually are patterned:
hist(rand, bins = 100)

# Create a fractal raster:
fractal <- fractalRast(elev, n = 2, dimension = c(2.1, 2.8))
plot(fractal)
hist(fractal)
```

```
}
```

`rSpatialDepRast,GRaster-method`

Create a random raster with or without spatial dependence

Description

`rSpatialDepRast()` creates a raster with random values in cells. Across the raster, values are approximately normally distributed, though a raster with a "true" normal distribution can be made with `rnormRast()`. Spatial dependence can be introduced, though all together the values will still be approximately normally distributed.

Usage

```
## S4 method for signature 'GRaster'
rSpatialDepRast(
  x,
  n = 1,
  mu = 0,
  sigma = 1,
  dist = 0,
  exponent = 1,
  delay = 0,
  seed = NULL
)
```

Arguments

`x`	A `GRaster`: The output will have the same extent and dimensions as this raster.
`n`	An integer: Number of rasters to generate.
`mu`, `sigma`	Numeric: Mean and sample standard deviation of output. If creating more than one raster, you can provide one value per raster. If there are fewer, they will be recycled.
`dist`	Numeric: Maximum distance of spatial autocorrelation (in map units–typically meters). Default is 0 (no spatial autocorrelation). If creating more than one raster, you can provide one value per raster. If there are fewer, values will be recycled.
`exponent`	Numeric > 0: Distance decay exponent. If creating more than one raster, you can provide one value per raster. If there are fewer, values will be recycled.
`delay`	Numeric >= 0: Values >0 force the distance decay of similarity to remain constant up to this distance. Beyond this distance, the decay exponent takes effect. Default is 0. If creating more than one raster, you can provide one value per raster. If there are fewer, values will be recycled.
`seed`	Numeric integer or `NULL`: Random seed. If `NULL`, then a random seed is used for each raster. If provided, there should be one seed value per raster.

Value

A GRaster.

See Also

rnormRast(), fractalRast(), runifRast(), **GRASS** manual page for module r.random.surface (see grassHelp("r.random.surface"))

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Elevation raster
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster:
elev <- fast(madElev)

### Create a raster with values drawn from a uniform distribution:
unif <- runifRast(elev)
plot(unif)

### Create a raster with values drawn from a normal distribution:
norms <- rnormRast(elev, n = 2, mu = c(5, 10), sigma = c(2, 1))
plot(norms)
hist(norms, bins = 100)

# Create a raster with random, seemingly normally-distributed values:
rand <- rSpatialDepRast(elev, dist = 1000)
plot(rand)

# Values appear normal on first inspection:
hist(rand)

# ... but actually are patterned:
hist(rand, bins = 100)

# Create a fractal raster:
fractal <- fractalRast(elev, n = 2, dimension = c(2.1, 2.8))
plot(fractal)
hist(fractal)

}
```

ruggedness,GRaster-method

Terrain ruggedness index

Description

For a given focal grid cell, the terrain ruggedness index (TRI) is calculated by taking the square root of the average of the squared difference between the focal cell's elevation and the elevations of the 8 surrounding cells, or

$$\sqrt{(\sum_{i=1}^{8}(m_i - m_0)^2/8)}$$

where m_0 is the elevation of the focal cell and m_i is the elevation of the *i*th grid cell.

Usage

```
## S4 method for signature 'GRaster'
ruggedness(x)
```

Arguments

x	A GRaster.

Value

A GRaster.

References

Riley, S.J., DeGloria, S.D., and Elliot, R. 1999. A terrain ruggedness index that quantifies topographic heterogeneity. *Intermountain Journal of Sciences* 5:23-27.

See Also

terrain(), wetness(), geomorphons()

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Elevation raster
madElev <- fastData("madElev")

# Convert to GRaster:
elev <- fast(madElev)
```

```
# Terrain ruggedness index:
tri <- ruggedness(elev)
plot(c(elev, tri))

# Topographic wetness index:
twi <- wetness(elev)
plot(c(elev, twi))

}
```

---

runifRast,GRaster-method

Create a raster with random values drawn from a uniform distribution

---

Description

`runifRast()` creates a raster with values drawn from a uniform (flat) distribution.

Usage

```
## S4 method for signature 'GRaster'
runifRast(x, n = 1, low = 0, high = 1, seed = NULL)
```

Arguments

`x`	A `GRaster`. The output will have the same extent and dimensions as this raster.
`n`	A numeric integer: Number of rasters to generate.
`low`, `high`	Numeric: Minimum and maximum values from which to select.
`seed`	Numeric integer vector or `NULL`: Random seed. If `NULL`, then a different seed will be generated by **GRASS**. Defining this is useful if you want to recreate rasters. If provided, there should be one seed value per raster.

Value

A `GRaster`.

See Also

`rnormRast()`, `rSpatialDepRast()`, `fractalRast()`, **GRASS** manual page for module `r.random.surface` (see `grassHelp("r.random.surface")`)

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Elevation raster
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster:
elev <- fast(madElev)

### Create a raster with values drawn from a uniform distribution:
unif <- runifRast(elev)
plot(unif)

### Create a raster with values drawn from a normal distribution:
norms <- rnormRast(elev, n = 2, mu = c(5, 10), sigma = c(2, 1))
plot(norms)
hist(norms, bins = 100)

# Create a raster with random, seemingly normally-distributed values:
rand <- rSpatialDepRast(elev, dist = 1000)
plot(rand)

# Values appear normal on first inspection:
hist(rand)

# ... but actually are patterned:
hist(rand, bins = 100)

# Create a fractal raster:
fractal <- fractalRast(elev, n = 2, dimension = c(2.1, 2.8))
plot(fractal)
hist(fractal)

}
```

`rvoronoi,GRaster-method`

Create a randomly-positioned tesselation

Description

This function partitions a region into Voronoi polygons that completely overlap it. Each polygon has a random center. The function is essentially a wrapper for `spatSample()` and `voronoi()`.

Usage

```
## S4 method for signature 'GRaster'
rvoronoi(x, size = 100, seed = NULL)

## S4 method for signature 'GVector'
rvoronoi(x, size = 100, seed = NULL)
```

Arguments

x	A GRaster or GVector used to constrain the location of random points used to create the tesselation.
size	Numeric integer or integer: Number of polygons.
seed	Numeric integer, integer, or NULL (default): Value used as a random seed. If NULL, a random seed will be generated by **GRASS**.

Value

A GVector.

Examples

```
if (grassStarted()) {

# Setup
library(sf)

# Example vectors
madDypsis <- fastData("madDypsis") # points
madCoast4 <- fastData("madCoast4") # polygons

# Convert sf vectors to GVectors
dypsis <- fast(madDypsis)
coast4 <- fast(madCoast4)
ant <- coast4[coast4$NAME_4 == "Antanambe"]

# Delaunay triangulation
dypsisDel <- delaunay(dypsis)
plot(dypsisDel)
plot(dypsis, pch = 1, col = "red", add = TRUE)

# Voronoi tessellation
vor <- voronoi(dypsis)
plot(vor)
plot(dypsis, pch = 1, col = "red", add = TRUE)

# Random Voronoi tessellation
rand <- rvoronoi(coast4, size = 100)
plot(rand)

}
```

sampleRast,GRaster-method

Randomly sample cells from a GRaster

Description

`sampleRast()` randomly samples cells from non-`NA` cells of a raster. The output will be a raster with selected non-`NA` cells, and all other cells set to `NA`. To generate random points, see `spatSample()`.

Usage

```
## S4 method for signature 'GRaster'
sampleRast(
  x,
  size,
  prop = FALSE,
  maskvalues = NA,
  updatevalue = NULL,
  test = FALSE,
  seed = NULL
)
```

Arguments

`x`	A `GRaster`.
`size`	Numeric: Number of cells or proportion of cells to select.
`prop`	Logical: If `TRUE`, the value of `size` will be interpreted as a proportion of cells. The default is `FALSE` (`size` is interpreted as the number of cells to select).
`maskvalues`	Numeric vector, including `NA`, or `NULL` (default): Values in the raster to select from. All others will be ignored. If this is `NULL`, then only non-`NA` cells will be selected for retention.
`updatevalue`	Numeric or `NULL` (default): Value to assign to masked cells. If `NULL`, then the values in the input raster are retained.
`test`	Logical: If `TRUE`, and `size` is greater than the number of non-`NA` cells in `x`, then fail. Testing this can take a long time for large rasters. The default is `FALSE`.
`seed`	`NULL` (default) or numeric: If `NULL`, then a random seed will be generated for the random number generator. Otherwise a seed can be provided.

Value

A `GRaster`.

See Also

`spatSample()`; `terra::spatSample()`, module `r.random` in **GRASS**

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Example data
madElev <- fastData("madElev") # raster

# Convert to GRasters and GVectors
elev <- fast(madElev)

### spatSample()

# Random points as data.frame or data.table:
randVals <- spatSample(elev, size = 20, values = TRUE)
randVals

# Random points as a points GVector:
randPoints <- spatSample(elev, size = 20, as.points = TRUE)
randPoints
plot(elev)
plot(randPoints, add = TRUE)

# Random points in a select area:
madCoast <- fastData("madCoast4") # vector
coast <- fast(madCoast)
ant <- coast[coast$NAME_4 == "Antanambe"] # subset

restrictedPoints <- spatSample(elev, size = 20, as.points = TRUE,
   strata = ant)

plot(elev)
plot(ant, add = TRUE)
plot(restrictedPoints, add = TRUE) # note 20 points for entire geometry

# Random points, one set per subgeometry:
stratifiedPoints <- spatSample(elev, size = 20, as.points = TRUE,
   strata = ant, byStratum = TRUE)

plot(elev)
plot(ant, add = TRUE)
plot(stratifiedPoints, pch = 21, bg = "red", add = TRUE) # note 20 points per subgeometry

# Random categories:
madCover <- fastData("madCover") # raster
cover <- fast(madCover)

randCover <- spatSample(cover, size = 20, values = TRUE,
     cat = TRUE, xy = TRUE)
randCover
```

```
### sampleRast()

# Random cells in non-NA cells:
rand <- sampleRast(elev, 10000)
plot(rand)
nonnacell(rand)

# Use custom values for the mask:
randCustomMask <- sampleRast(elev, 10000, maskvalues = 1:20)
plot(randCustomMask)

# Force selected values to a custom value:
randCustomUpdate <- sampleRast(elev, 10000, updatevalue = 7)
plot(randCustomUpdate)

# Custom values for mask and set selected cells to custom value:
randAll <- sampleRast(elev, 10000, maskvalues = 1:20, updatevalue = 7)
plot(randAll)

}
```

---

scale,GRaster-method *Center and scale a GRaster, or the opposite*

Description

`scale()` and `scalepop()` center and scale layers in a `GRaster` by subtracting from each raster its mean value (centering), then dividing by its standard deviation (scaling). This is useful for using the raster in a linear model, for example, because unscaled predictors can lead to numerical instability. The `scale()` function uses the sample standard deviation, and the `scalepop()` function uses the population standard deviation. For even moderately-sized rasters, the difference between these two is negligible, but the `scalepop()` function can be much faster than the `scale()` function.

The `unscale()` function does the opposite of `scale()` and `scalepop()`: it multiples each layer by a value (presumably, its standard deviation), and adds another value (presumably, its mean).

Usage

```
## S4 method for signature 'GRaster'
scale(x, center = TRUE, scale = TRUE)

## S4 method for signature 'GRaster'
scalepop(x, center = TRUE, scale = TRUE)

## S4 method for signature 'GRaster'
unscale(x, center = NULL, scale = NULL)
```

Arguments

x	A GRaster.
center	Value depends on the function: • scale(): Logical: If TRUE (default), subtract from each raster layer its mean. • unscale(): Numeric vector or NULL (default): This can be a single value, which will be recycled if there is more than one layer in the raster, or one value per raster layer. If a value is NA, then no un-centering will be performed on the relevant raster layer. If NULL, then no un-centering is done.
scale	Value depends on the function: • scale(): Logical: If TRUE (default), divide each layer by its standard deviation. • unscale(): Numeric vector or NULL (default): This can be a single value, which will be recycled if there is more than one layer in the raster, or one value per raster layer. If a value is NA, then no unscaling will be done on the relevant raster layer. If NULL, then no un-scaling is done.

Value

All functions return a GRaster. The output of scale() and scalepop() will have two attributes, "center" and "scale", which have the means and standard deviations of the original rasters (if center and scale are TRUE, otherwise, they will be NA). These can be obtained using attributes(output_raster)$center and attributes(output_raster)$scale.

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Climate rasters:
madChelsa <- fastData("madChelsa")

# Convert to GRasters:
chelsa <- fast(madChelsa)

### Center and scale rasters
# Scale with using sample SD:
chScaled <- scale(chelsa)
chScaled

# Scale with using population SD:
chScaledPop <- scalepop(chelsa)
chScaledPop

# Means are very close to 0 and SDs to 1:
global(chScaled, c("mean", "sd", "min", "max"))
global(chScaledPop, c("mean", "sd", "min", "max"))
```

```
# Get original means and sd's:
centers <- attributes(chScaled)$center
scales <- attributes(chScaled)$scale
centers
scales

### Unscale rasters:
chUnscaled <- unscale(chScaled, center = centers, scale = scales)

# Means and SD are returned to original values:
global(chUnscaled, c("mean", "sd", "min", "max")) # unscaled
global(chelsa, c("mean", "sd", "min", "max")) # original

}
```

`segregate,GRaster-method`

Create one GRaster layer per unique value in a GRaster

Description

This function creates a multi-layered `GRaster` for every unique values in an input `GRaster`. By default, the output will have a value of 1 wherever the input has the given value, and 0 elsewhere. This is useful for creating dummy variable `GRaster` layers for use with models that have factors, especially if the input `GRaster` is categorical. Note that the `predict()` function in **fasterRaster** usually does not need this treatment of `GRaster`s since it can handle categorical rasters already.

Usage

```
## S4 method for signature 'GRaster'
segregate(x, classes = NULL, keep = FALSE, other = 0, bins = 100, digits = 3)
```

Arguments

`x`	A `GRaster`.
`classes`	Either `NULL` (default) or a character vector with category labels for which to create outputs. If the input is not a categorical/factor or `integer` `GRaster`, this is ignored.
`keep`	Logical: If `FALSE` (default), then the original value in the input `GRaster` will be retained in the each of the output `GRaster` layers wherever the input had the respective value. Other cells will be assigned a value of `other`.
`other`	Numeric or `NA`: Value to assign to cells that do not have the target value.
`bins`	Numeric: Number of bins in which to put values. This is only used for `GRaster`s that are not categorical/factor rasters or `integer` rasters.
`digits`	Numeric: Number of digits to which to round input if it is a `numeric` or `double` `GRaster` (see `vignettes("GRasters", package = "fasterRaster")`).

Value

If the input `x` is a single-layered `GRaster`, the output will be a multi-layered `GRaster` with one layer per value in the input, or one layer per values in `classes`. If the input is a multi-layered `GRaster`, the output will be a `list` of multi-layered `GRaster`s.

See Also

`terra::segregate()`

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Elevation and land cover raster
madElev <- fastData("madElev") # integer raster
madCover <- fastData("madCover") # categorical raster

# Convert to GRasters
elev <- fast(madElev)
cover <- fast(madCover)

# Subset elevation raster to just a few values to make example faster:
elevSubset <- elev[elev <= 3]
segregate(elevSubset)
segregate(elevSubset, keep = TRUE, other = -1)

# Segregate the factor raster
segregate(cover)

classes <- c("Grassland with mosaic forest", "Mosaic cropland/vegetation")
seg <- segregate(cover, classes = classes)
plot(seg)

}
```

---

selectRange,GRaster-method

Select values from rasters in a stack based on values in another raster

---

Description

`selectRange()` selects values from `GRaster`s in a "stack" based on the values in another "selection" raster. For example, if the stack has three layers (call them A, B, and C), the "selection" raster could have values of 1, 2, or 3 in each cell. The raster that is returned will have values from A wherever the selection raster is 1, B from where it is 2, and C from where it is 3.

Usage

```
## S4 method for signature 'GRaster'
selectRange(x, y)
```

Arguments

x	A GRaster, typically with more than one layer.
y	A GRaster with integer values. The raster will be rounded if it does not. The values are typically between 1 and the number of layers in x, but wherever they are outside this range, the returned raster will have NA values.

Value

A GRaster.

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster
elev <- fast(madElev)

# Make a stack of various versions of "elev" from which to select from:
x <- c(elev, 10 * elev, ln(elev), -1 * elev)
x

# Make a layer with random numbers between 1 and 4:
fun <- "= round(rand(0.5, 4.5))"
y <- app(elev, fun = fun)

selected <- selectRange(x, y)

}
```

---

seqToSQL *Format a numeric series into an SQL value call*

Description

This function takes as its argument a vector of integers or numeric values, and converts sequential runs to a range while keeping non-sequential values as-is. For example, `c(1, 5, 6, 7, 8, 9, 15, 16, 20)` becomes `"1,5-9,15-16,20"`. This reduces the number of characters necessary to supply to a SQL condition. This function is mainly of use to developers.

Usage

```
seqToSQL(x, maxChar = 29900, sort = TRUE)
```

Arguments

x	A vector of numerical values. The vector should be sorted from lowers to highest for the most efficient "compression" of sequential ranges. Values will be coerced to class integer.
maxChar	Integer or numeric: Maximum number of characters to include in the output. If the output has more than this number of characters, the remainder is dropped, and the trim attribute of the output is set to TRUE. The default is 29900, which is the maximum length of an SQL statement that **GRASS** seems to be able to handle (minus a safety margin).
sort	Logical: If TRUE (default), sort x before converting to SQL. This can reduce the length of the output.

Value

A character string. The string has three attributes. The trim attribute is TRUE or FALSE, depending on whether maxChar was reached or not (and subsequent numbers dropped from the string). The lastIndex attribute is the last index of x that was processed (i.e., the index of the last value in the output), and the number of values represented by the output.

Examples

```
x <- 1:5
seqToSQL(x)

x <- c(1:5, 7)
seqToSQL(x)

x <- c(1:5, 7, 15:16)
y <- c(1:5, 7, 15:16, 20)
seqToSQL(x)
seqToSQL(y)

seqToSQL(x, maxChar = 5)
seqToSQL(y, maxChar = 8)

seqToSQL(10:1, sort = FALSE)
seqToSQL(10:1, sort = TRUE)
```

`simplifyGeom,GVector-method`
Simplify the geometry of a vector

Description

`simplifyGeom()` reduces the number of vertices used to represent a vector (i.e., to save memory or disk space). There are several methods available.

Usage

```
## S4 method for signature 'GVector'
simplifyGeom(x, tolerance = NULL, method = "VR", prop = 0.5)
```

Arguments

`x`	A `GVector`.
`tolerance`	Numeric >= 0: Threshold distance in map units (degrees for unprojected, usually meters for projected). If `NULL`, then 2% of the minimum of the x-, y-, and z-extent will be used.
`method`	Character: Method used to reduce the number of vertices. Partial matching is used, and case does not matter: • `"VR"`: Vertex reduction (default, simplest): If two points p1 and p2 on the same line are closer than the threshold, remove p2. The `tolerance` argument represents this threshold distance. • `"DP"`: Douglas-Peucker (AKA Ramer-Douglas-Peucker) algorithm: Simply stated, for points p1, p2, and p3 on a line, this method constructs a line segment between p1 and p3. If p2 is closer than the threshold to the line segment, it is removed. In this example, the `tolerance` argument refers to the maximum distance between p2 and the line segment. • `"DPR"`: Douglas-Peucker algorithm with reduction: As the Douglas-Pueker method, but each geometry is thinned so that in the end it has only a given proportion of the starting number of points. The `prop` argument refers to this proportion of remaining points. • `"RW`: Reumann-Witkam algorithm: For points p1, p2, p3, and p4 on a line, constructs two line segments parallel to the line segment defined by p1 and p4. These are placed `tolerance` distance one either side of the p1-p4 line segment. If the line segment p1-p2 or p3-p4 falls entirely within the bounds of the two outer parallel segments, p2 and p3 are removed, leaving just p1 and p4.
`prop`	Positive value between 0 and 1: Proportion of points that will be retained for each geometry when the Douglas-Peucker algorithm with reduction is applied (ignored otherwise). Default is 0.5 (retain 50% of vertices).

Value

A GVector.

See Also

smoothGeom(), geometry cleaning, terra::simplifyGeom(), **GRASS** manual page for module v.generalize (see grassHelp("v.generalize"))

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Example data
madRivers <- fastData("madRivers")
rivers <- fast(madRivers)
soam <- rivers[rivers$NAM == "SOAMIANINA"] # select one river for illustration

### Simplify geometry (remove nodes)

vr <- simplifyGeom(soam, tolerance = 2000)
dp <- simplifyGeom(soam, tolerance = 2000, method = "dp")
dpr <- simplifyGeom(soam, tolerance = 2000, method = "dpr", prop = 0.5)
rw <- simplifyGeom(soam, tolerance = 2000, method = "rw")

plot(soam, col = "black", lwd = 3)
plot(vr, col = "blue", add = TRUE)
plot(dp, col = "red", add = TRUE)
plot(dpr, col = "chartreuse", add = TRUE)
plot(rw, col = "orange", add = TRUE)

legend("bottom",
   xpd = NA,
   legend = c(
   "Original",
      "Vertex reduction",
      "Douglas-Peucker",
      "Douglas-Peucker reduction",
      "Reumann-Witkam"
 ),
 col = c("black", "blue", "red", "chartreuse", "orange"),
 lwd = c(3, 1, 1, 1, 1)
)

### Smooth geometry

hermite <- smoothGeom(soam, dist = 2000, angle = 3)
chaiken <- smoothGeom(soam, method = "Chaiken", dist = 2000)
```

```
plot(soam, col = "black", lwd = 2)
plot(hermite, col = "blue", add = TRUE)
plot(chaiken, col = "red", add = TRUE)

legend("bottom",
   xpd = NA,
   legend = c(
   "Original",
      "Hermite",
      "Chaiken"
 ),
 col = c("black", "blue", "red"),
 lwd = c(2, 1, 1, 1, 1)
)

### Clean geometry

# Has no effect on this vector!
noDangs <- removeDangles(soam, tolerance = 10000)

plot(soam, col = "black", lwd = 2)
plot(noDangs, col = "red", add = TRUE)

legend("bottom",
   xpd = NA,
   legend = c(
   "Original",
      "No dangles"
 ),
 lwd = c(2, 1),
 col = c("black", "red")
)

}
```

`sineRast,GRaster-method` *Sine wave rasters*

Description

This function creates one or more rasters with sine waves in the north-south and east-west directions.

Usage

```
## S4 method for signature 'GRaster'
sineRast(
  x,
  ns = 1,
  ew = 1,
```

```
  nsOffset = 0,
  ewOffset = 0,
  nsAmp = 1,
  ewAmp = 1,
  combos = FALSE,
  mask = NULL,
  verbose = FALSE
)
```

Arguments

x	A GRaster.
ns, ew	Numeric: Number of complete sine waves (i.e., wavelengths) in the north-south and east-west directions. A wavelength of 1 creates a "full" sine wave (e.g., starting at 0 at one end and ending at 0 at the other). A wavelength of 2 would create two such waves, and so on. A value of 0 creates no waves in the given direction (i.e., each row or column has constant values). The default value is 1.
nsOffset, ewOffset	Numeric: Offset of the sine waves from the edges of the raster, expressed as a proportion of the length of the raster. The default is 0, so the values of the outermost cells will be close to 0 (but not exactly 0 because centers of cells at the raster edges are not on the actual edge). If an offset value is 0.2, for example, then it will be pushed "inward" toward the middle of the raster by 20% of the raster's extent.
nsAmp, ewAmp	Numeric: Amplitude (minimum and maximum of the sine wave) in the north-south and east-west directions. The default is 1. Note that when north-south and east-west waves are created (i.e., ns and ew are both > 0), the effective amplitude is halved so that the sum is equal to nsAmp + ewAmp.
combos	Logical: If TRUE (default), create sine rasters using all possible combinations of values of ns, ew, nsOffset, ewOffset, and amp. If FALSE, you can only supply either one value per parameter, or the same number of values per parameter. In this latter case, one raster will be created per pairwise set of the unique parameters. For example, you could specify 3 values for ns and either one or three values for any of the other parameters, and three rasters would be created.
mask	Either NULL (default), or a GRaster or GVector: Used as a mask for the output. If this is a GVector, then only cells that are not NA in the mask have values. If this is a GVector, then only cells that overlap the vector have values assigned. All other values will be NA.
verbose	Logical: If TRUE, display progress.

Value

A GRaster.

Examples

```
if (grassStarted()) {
```

```
# Setup
library(sf)
library(terra)

# Elevation raster
madElev <- fastData("madElev")

# Convert to GRaster:
elev <- fast(madElev)

### Simple sine waves:
waves <- sineRast(elev, ns = 2, ew = 1)
plot(waves)

### Sine waves with different amplitudes:
amps <- sineRast(elev, nsAmp = c(1, 5), ewAmp = c(1, 5))
amps

### Sine waves with and without north-south offset:
noOffsets <- sineRast(elev, ns = 1, ew = 1)
offsets <- sineRast(elev, ns = 1, ew = 1, nsOffset = 0.25)
offs <- c(noOffsets, offsets)
names(offs) <- c("no offset", "offset")
plot(offs)

### Masking:
madCoast4 <- fastData("madCoast4")
coast4 <- fast(madCoast4, verbose = FALSE)

masked <- sineRast(elev, mask = coast4)
plot(masked)

### Multiple sine waves (multiple rasters):
mults <- sineRast(elev, ns = 1:2, ew = 1:2)
combos <- sineRast(elev, ns = 1:2, ew = 1:2, combos = TRUE)
plot(mults)
plot(combos)

}
```

---

smoothGeom,GVector-method

Smooth the geometry of a vector

Description

`smoothGeom()` makes line segments of a vector appear less angular.

Usage

```
## S4 method for signature 'GVector'
smoothGeom(x, method = "Hermite", dist = NULL, angle = 3)
```

Arguments

x	A GVector.
method	Character: Method used to smooth line segments. Partial matching is used, and case does not matter: • "Hermite": Hermite interpolation (default): Guarantees that the output vector always passes through the original points. This method adds points (possibly many) by constructing cubic splines with points approximately dist apart. The number of points can be reduced by specifying a smaller value of angle, which specifies the minimum angle between two successive line segments. • "Chaiken": Chaiken's algorithm: Guarantees that the new vector always touches the midpoint of each original line segment. The points on the new line are at least dist apart.
dist	Numeric > 0 of NULL (default): Minimum distance (see method). Units are in map units. If NULL, then 2% of the minimum of the x-, y-, and z-extent will be used.
angle	Numeric > 0: Maximum angle for the Hermite algorithm. Default is 3.

Value

A GVector.

See Also

simplifyGeom(), terra::simplifyGeom(), geometry cleaning, **GRASS** manual page for module v.generalize (see grassHelp("v.generalize"))

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Example data
madRivers <- fastData("madRivers")
rivers <- fast(madRivers)
soam <- rivers[rivers$NAM == "SOAMIANINA"] # select one river for illustration

### Simplify geometry (remove nodes)

vr <- simplifyGeom(soam, tolerance = 2000)
dp <- simplifyGeom(soam, tolerance = 2000, method = "dp")
```

```
dpr <- simplifyGeom(soam, tolerance = 2000, method = "dpr", prop = 0.5)
rw <- simplifyGeom(soam, tolerance = 2000, method = "rw")

plot(soam, col = "black", lwd = 3)
plot(vr, col = "blue", add = TRUE)
plot(dp, col = "red", add = TRUE)
plot(dpr, col = "chartreuse", add = TRUE)
plot(rw, col = "orange", add = TRUE)

legend("bottom",
   xpd = NA,
   legend = c(
   "Original",
      "Vertex reduction",
      "Douglas-Peucker",
      "Douglas-Peucker reduction",
      "Reumann-Witkam"
 ),
 col = c("black", "blue", "red", "chartreuse", "orange"),
 lwd = c(3, 1, 1, 1, 1)
)

### Smooth geometry

hermite <- smoothGeom(soam, dist = 2000, angle = 3)
chaiken <- smoothGeom(soam, method = "Chaiken", dist = 2000)

plot(soam, col = "black", lwd = 2)
plot(hermite, col = "blue", add = TRUE)
plot(chaiken, col = "red", add = TRUE)

legend("bottom",
   xpd = NA,
   legend = c(
   "Original",
      "Hermite",
      "Chaiken"
 ),
 col = c("black", "blue", "red"),
 lwd = c(2, 1, 1, 1, 1)
)

### Clean geometry

# Has no effect on this vector!
noDangs <- removeDangles(soam, tolerance = 10000)

plot(soam, col = "black", lwd = 2)
plot(noDangs, col = "red", add = TRUE)

legend("bottom",
   xpd = NA,
   legend = c(
```

```
    "Original",
       "No dangles"
  ),
  lwd = c(2, 1),
  col = c("black", "red")
)

}
```

---

sources,GRaster-method

Name of a raster or vector in a GRASS session

Description

sources() retrieves the name of a raster or vector in **GRASS**. GRasters and GVectors are actually pointers to objects stored in a **GRASS** database. When using **fasterRaster** functions on rasters and vectors, the commands are translated into **GRASS** commands and executed on the objects named in the pointers. These objects use a "source" (which is really a filename) to refer to the **GRASS** objects. This function is mostly of use to developers.

Usage

```
## S4 method for signature 'GRaster'
sources(x)

## S4 method for signature 'GVector'
sources(x)

## S4 method for signature 'character'
sources(x)
```

Arguments

x	Either a GSpatial object or one that inherits from it (i.e., a GRaster or GVector), *or* a character. If a character, then the character itself is returned.

Value

Character.

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
```

```
madElev <- fastData("madElev")
madForest2000 <- fastData("madForest2000")

# Convert SpatRasters to GRasters
elev <- fast(madElev)
forest <- fast(madForest2000)

### GRaster properties

# plotting
plot(elev)

# dimensions
dim(elev) # rows, columns, depths, layers
nrow(elev) # rows
ncol(elev) # columns
ndepth(elev) # depths
nlyr(elev) # layers

res(elev) # resolution (2D)
res3d(elev) # resolution (3D)
zres(elev) # vertical resolution
xres(elev) # vertical resolution
yres(elev) # vertical resolution
zres(elev) # vertical resolution (NA because this is a 2D GRaster)

# cell counts
ncell(elev) # cells
ncell3d(elev) # cells (3D rasters only)

# number of NA and non-NA cells
nacell(elev)
nonnacell(elev)

# topology
topology(elev) # number of dimensions
is.2d(elev) # is it 2-dimensional?
is.3d(elev) # is it 3-dimensional?

minmax(elev) # min/max values

# "names" of the object
names(elev)

# coordinate reference system
crs(elev)
st_crs(elev)
coordRef(elev)

# extent (bounding box)
ext(elev)

# vertical extent (not defined for this raster)
```

```
zext(elev)

# data type
datatype(elev) # fasterRaster type
datatype(elev, "GRASS") # GRASS type
datatype(elev, "terra") # terra type
datatype(elev, "GDAL") # GDAL type

is.integer(elev)
is.float(elev)
is.double(elev)
is.factor(elev)

# convert data type
as.int(elev) # integer; note that "elev" is already of type "integer"
as.float(elev) # floating-precision
as.doub(elev) # double-precision

# assigning
pie <- elev
pie[] <- pi # assign all cells to the value of pi
pie

# concatenating multiple GRasters
rasts <- c(elev, forest)
rasts

# subsetting
rasts[[1]]
rasts[["madForest2000"]]

# replacing
rasts[[2]] <- 2 * forest
rasts

# adding layers
rasts[[3]] <- elev > 500 # add a layer
rasts <- c(rasts, sqrt(elev)) # add another
add(rasts) <- ln(elev)
rasts

# names
names(rasts)
names(rasts) <- c("elev_meters", "2_x_forest", "high_elevation", "sqrt_elev", "ln_elev")
rasts

# remove a layer
rasts[["2_x_forest"]] <- NULL
rasts

# number of layers
nlyr(rasts)
```

```
# correlation and covariance matrices
madLANDSAT <- fastData("madLANDSAT")
landsat <- fast(madLANDSAT) # projects matrix
layerCor(landsat) # correlation
layerCor(landsat, fun = 'cov') # covariance

}
```

---

spatSample,GRaster-method

Sample random points from a GRaster or GVector

Description

spatSample() randomly locates points across a GRaster or GVector. It can return a GVector, the coordinates, values associated with the points, or all of these. If you want to generate a raster with randomly-sampled cells, see sampleRast().

Usage

```
## S4 method for signature 'GRaster'
spatSample(
  x,
  size,
  as.points = FALSE,
  values = TRUE,
  cats = TRUE,
  xy = FALSE,
  strata = NULL,
  byStratum = FALSE,
  zlim = NULL,
  seed = NULL,
  verbose = FALSE
)

## S4 method for signature 'GVector'
spatSample(
  x,
  size,
  as.points = FALSE,
  values = TRUE,
  xy = FALSE,
  byStratum = FALSE,
  zlim = NULL,
  seed = NULL
)
```

Arguments

x	A GRaster or GVector.
size	Numeric value > 0: Number of points to create.
as.points	Logical: If FALSE (default), the output is a data.frame or data.table. If TRUE, the output is a "points" GVector.
values	Logical: If TRUE (default), values of the GRaster at points are returned.
cats	Logical: If TRUE (default) and the GRaster is categorical, then return the category label of each cell. If values is also TRUE, then the cell value will also be returned.
xy	Logical: If TRUE, return the longitude and latitude of each point. Default is FALSE.
strata	Either NULL (default), or a GVector defining strata. If supplied, the size argument will be interpreted as number of points to place per geometry in strata. Note that using strata can dramatically slow the process.
byStratum	Logical: If FALSE (default), then size number of points will be placed within the entire area delineated by strata. If TRUE, then size points will be placed within each subgeometry of strata.
zlim	Either NULL (default), or a vector of two numbers defining the lower and upper altitudinal bounds of coordinates. This cannot be combined with values = TRUE or cats = TRUE.
seed	Either NULL (default) or an integer: Random number seed. If this is NULL, the a seed will be set randomly. Values will be rounded to the nearest integer.
verbose	Logical: If TRUE, display progress. Default is FALSE.

Value

A data.frame, data.table, or GVector.

See Also

sampleRast(), terra::spatSample(), module v.random in **GRASS**

Examples

```
if (grassStarted()) {

# Setup
library(sf)
library(terra)

# Example data
madElev <- fastData("madElev") # raster

# Convert to GRasters and GVectors
elev <- fast(madElev)

### spatSample()
```

```
# Random points as data.frame or data.table:
randVals <- spatSample(elev, size = 20, values = TRUE)
randVals

# Random points as a points GVector:
randPoints <- spatSample(elev, size = 20, as.points = TRUE)
randPoints
plot(elev)
plot(randPoints, add = TRUE)

# Random points in a select area:
madCoast <- fastData("madCoast4") # vector
coast <- fast(madCoast)
ant <- coast[coast$NAME_4 == "Antanambe"] # subset

restrictedPoints <- spatSample(elev, size = 20, as.points = TRUE,
   strata = ant)

plot(elev)
plot(ant, add = TRUE)
plot(restrictedPoints, add = TRUE) # note 20 points for entire geometry

# Random points, one set per subgeometry:
stratifiedPoints <- spatSample(elev, size = 20, as.points = TRUE,
   strata = ant, byStratum = TRUE)

plot(elev)
plot(ant, add = TRUE)
plot(stratifiedPoints, pch = 21, bg = "red", add = TRUE) # note 20 points per subgeometry

# Random categories:
madCover <- fastData("madCover") # raster
cover <- fast(madCover)

randCover <- spatSample(cover, size = 20, values = TRUE,
     cat = TRUE, xy = TRUE)
randCover

### sampleRast()

# Random cells in non-NA cells:
rand <- sampleRast(elev, 10000)
plot(rand)
nonnacell(rand)

# Use custom values for the mask:
randCustomMask <- sampleRast(elev, 10000, maskvalues = 1:20)
plot(randCustomMask)

# Force selected values to a custom value:
randCustomUpdate <- sampleRast(elev, 10000, updatevalue = 7)
plot(randCustomUpdate)
```

```
# Custom values for mask and set selected cells to custom value:
randAll <- sampleRast(elev, 10000, maskvalues = 1:20, updatevalue = 7)
plot(randAll)

}
```

`streams,GRaster-method`

Create stream network

Description

This function estimates the course of streams and rivers from an elevation raster. It is based on the **GRASS** module `r.stream.extract`, where more details can be found (see `grassHelp("r.stream.extract")`)

Usage

```
## S4 method for signature 'GRaster'
streams(
  x,
  accumulation = NULL,
  depression = NULL,
  flowThreshold = 1,
  dirThreshold = 1,
  montgomery = 0,
  minLength = 1
)
```

Arguments

`x`	A `GRaster` representing elevation.
`accumulation`	Either `NULL` (default) or a raster representing flow accumulation. If not supplied, an accumulation will created internally. You can generate an accumulation raster using `flow()`.
`depression`	Either `NULL` (default) or a `GRaster` representing depressions (areas from which streams will not flow out of).
`flowThreshold`	Numeric > 0: Minimum threshold for a stream to be generated. The default is 1, which is not necessarily a reasonable value.
`dirThreshold`	Numeric (default is `Inf`): When flow exceeds this threshold, its direction is estimated using a single-flow direction algorithm. Below this threshold, a multi-direction flow model is used. This is the `d8cut` parameter in `r.stream.extract`, and it is only used if `accumulation` is `NULL`. The default is 1, which is not necessarily a reasonable value.
`montgomery`	Numeric: The "Montgomery" exponent for slope, multiplied by accumulation as per `accumulation * slope^montgomery`. This value is then compared to the threshold to determine if it is sufficient. The default is 0 (i.e., no slope scaling).

minLength Numeric: First-order streams less than this length are removed (units in cells). Default is 0 (no removal).

Value

A GRaster.

See Also

flow(), flowPath(), **GRASS** manual for module r.stream.extract (see grassHelp("r.stream.extract"))

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster
elev <- fast(madElev)

# Calculate stream channels
streams <- streams(elev)
plot(streams)

}
```

---

stretch,GRaster-method

Rescale values in a GRaster

Description

stretch() rescales the values in a GRaster. All values can be rescaled, or just values in a user-defined range. This range can be given by specifying either the lower and upper bounds of the range using smin and smax, and/or by the quantiles (across all cells of the raster) using minq and maxq.

Usage

```
## S4 method for signature 'GRaster'
stretch(x, minv = 0, maxv = 255, minq = 0, maxq = 1, smin = NA, smax = NA)
```

Arguments

x	A GRaster.
minv, maxv	Numeric: Minimum and maximum values to which to rescale values.
minq, maxq	Numeric: Specifies range of values to rescale, given by their quantiles. The default is to stretch all values (the 0th and 100th quantiles). One or both are ignored if smin and/or smax are provided.
smin, smax	Numeric or NA: Specifies range of values to rescale. If NA (default), then all values are rescaled.

Value

A GRaster.

See Also

terra::stretch() and module r.rescale in **GRASS** (not used on this function)

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster
elev <- fast(madElev)

### Stretch based on user-defined range

#  fasterRaster
fr <- stretch(elev, smin=1, smax=100)
fr

# terra
tr <- stretch(madElev, smin = 1, smax = 100)
tr

# Compare fasterRaster to terra output
fr <- rast(fr)
fr <- extend(fr, tr)
fr - tr

### Stretch values in a certain quantile range

#  fasterRaster
fr <- stretch(elev, minq = 0.25, maxq = 0.75)
fr
```

```
# terra
tr <- stretch(madElev, minq = 0.25, maxq = 0.75)
tr

# Compare fasterRaster to terra output
fr <- rast(fr)
fr <- extend(fr, tr)
fr - tr

}
```

---

subset,GRaster-method *Subset layers from a GRaster, or specific rows from a GVector*

Description

subset() can be used to subset or remove one or more layers from a GRaster. It can also be used to subset or remove rows from a GVector with a data table.

Usage

```
## S4 method for signature 'GRaster'
subset(x, subset, negate = FALSE)

## S4 method for signature 'GVector'
subset(x, subset, negate = FALSE)
```

Arguments

x	A GRaster or GVector.
subset	Numeric integer, integer, logical, or character: Indicates the layer(s) of a GRaster to subset, or the rows(s) of a GVector to return.
negate	Logical: If TRUE, all layers or rows in subset will be *removed* from the output. Default is FALSE.

Value

A GRaster or GVector.

See Also

[[, [

Examples

```
if (grassStarted()) {

# Setup
library(terra)

### GRasters

# Example data
madElev <- fastData("madElev") # elevation raster
madForest2000 <- fastData("madForest2000") # forest raster
madForest2014 <- fastData("madForest2014") # forest raster

# Convert SpatRasters to GRasters
elev <- fast(madElev)
forest2000 <- fast(madForest2000)
forest2014 <- fast(madForest2014)

# Re-assigning values of a GRaster
constant <- elev
constant[] <- pi
names(constant) <- "pi_raster"
constant

# Re-assigning specific values of a raster
replace <- elev
replace[replace == 1] <- -20
replace

# Subsetting specific values of a raster based on another raster
elevInForest <- elev[forest2000 == 1]
plot(c(elev, forest2000, elevInForest), nr = 1)

# Adding and replacing layers of a GRaster
rasts <- c(elev, constant, forest2000)

# Combine with another layer:
add(rasts) <- forest2014 # one way
rasts

rasts <- c(rasts, forest2014) # another way

### Subsetting GRaster layers

# Subset:
rasts <- c(elev, forest2000, forest2014)
rasts[[2:3]]
subset(rasts, 2:3)
subset(rasts, c("madForest2000", "madElev"))
rasts[[c("madForest2000", "madElev")]]
rasts$madForest2000
```

```
# Get every other layer:
rasts[[c(FALSE, TRUE)]]

### Replacing layers of a GRaster

# Replace a layer
logElev <- log(elev)
names(logElev) <- "logElev"
rasts$madForest2014 <- logElev
rasts

# Replace a layer:
rasts[[3]] <- forest2000
rasts

### GVectors

# example data
madDypsis <- fastData("madDypsis") # vector of points

# Convert SpatVector to GVector
dypsis <- fast(madDypsis)

### Retrieving GVector columns

dypsis$species # Returns the column

dypsis[[c("year", "species")]] # Returns a GRaster with these columns
dypsis[ , c("year", "species")] # Same as above

### Subsetting GVector geometries

# Subset first three geometries
dypsis[1:3]
dypsis[1:3, "species"]

# Get geometries by data table condition
dypsis[dypsis$species == "Dypsis betsimisarakae"]

### (Re)assigning GVector column values

# New column
dypsis$pi <- pi

# Re-assign values
dypsis$pi <- "pie"

# Re-assign specific values
dypsis$institutionCode[dypsis$institutionCode == "MO"] <-
   "Missouri Botanical Garden"

}
```

subst,GRaster-method *Replace a specific value(s) in a GRaster*

Description

This function replaces one or more user-specified values in a raster with other values. See classify() for replacing ranges of values.

Usage

```
## S4 method for signature 'GRaster'
subst(x, from, to, others = NULL, warn = TRUE)
```

Arguments

x	A GRaster.
from, to	Vectors of numeric or character values. The value(s) in from will be replaced with the value(s) in to. They must be the same length, or, if you supply a single value for to, then all values in from will be converted to the same value of to. Numeric/integer or character vectors can be used: • from and to are numeric or integer vectors: Values in from will be replaced by their corresponding value in to. • from and to are character vectors: You can use a character vector for from and to. In this case, the input raster must be a factor (categorical) raster or an integer raster. If from is a character vector, these levels(categories) will be replaced by the levels in to. This can add levels to a GRaster to has labels that do not match any existing labels in from. • from is a character vector and to is an integer vector: Cells in x that correspond to the given label will have their values replaced by the corresponding value in to, and be matched the the corresponding label. If no label corresponds to the new value, a new level will be created. The input must be a categorical raster. • from is an integer vector and to is a character vector: Cells in x that have a value in from will be replaced by values that match the labels in to. If the input raster does not have a value that corresponds to the given label in y, then a new value will be created.
others	NULL (default), NA, numeric, or a character: • NULL (default): Values that do not appear in from will be unchanged. • NA: Values that do not appear in from will be set to NA. • Character: Cells in x that do not appear in to will be assigned to this level. In this case, x must be a categorical (factor) raster.
warn	Logical: If TRUE (default), display a warning when new levels are created.

Value

A GRaster.

See Also

terra::subst(), classify()

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")
madCover <- fastData("madCover")

### Substitution within an integer/numeric raster

# Convert SpatRaster to GRaster
elev <- fast(madElev)

# Simple substitution of one value, keeping all other values
newElev <- elev
newElev[newElev == 100] <- -100
newElev[newElev > 500] <- 500
hist(newElev)

# Simple substitution of one value, keeping all other values
substituted <- subst(elev, from = 300, to = -300)
substituteds <- c(elev, substituted)
names(substituteds) <- c("original", "substituted")
plot(substituteds)

# Simple substitution of three values, keeping all other values
substituted <- subst(elev, from = c(299, 300, 301), to = c(-699, -600, -601))
substituteds <- c(elev, substituted)
names(substituteds) <- c("original", "substituted")
plot(substituteds)

# Simple substitution of three values to one other value, retaining remainder
substituted <- subst(elev, from = c(299, 300, 301), to = -1000)
substituteds <- c(elev, substituted)
names(substituteds) <- c("original", "substituted")
plot(substituteds)

# Simple substitution of one value, setting all other values to 100
substituted <- subst(elev, from = 300, to = -300, others = 100)
substituteds <- c(elev, substituted)
names(substituteds) <- c("original", "substituted")
plot(substituteds)

### Substitution within a factor/categorical raster

# Convert a SpatRaster to a GRaster:
```

```
cover <- fast(madCover)

cover <- droplevels(cover) # remove unused levels
levels(cover) # levels of "cover"

# Substitute using level name, replace with EXISTING level label
from <- "Mosaic cropland/vegetation"
to <- "Mosaic crops"
categ <- subst(cover, from = from, to = to)
freq(cover) # original frequencies of each land cover class
freq(categ) # note change in frequency of "from" and "to" categories
plot(c(cover, categ))

# Substitute using level name, replace with NEW level label
from <- c("Mosaic crops", "Mosaic cropland/vegetation")
to <- c("Mixed cropland")
categ <- subst(cover, from = from, to = to)
freq(cover) # original frequencies of each land cover class
freq(categ) # note change in frequency of "from" and "to" categories
plot(c(cover, categ))

# Substitute using level name, replace with NEW level label
from <- c("Mosaic crops", "Mosaic cropland/vegetation")
to <- c("Mixed cropland", "Mixed cropland/vegetation")
categ <- subst(cover, from = from, to = to)
freq(cover) # original frequencies of each land cover class
freq(categ) # note change in frequency of "from" and "to" categories
plot(c(cover, categ))

# Substitute using level name, replace with VALUE of an existing label
from <- c("Mosaic crops", "Mosaic cropland/vegetation")
to <- 120
categ <- subst(cover, from = from, to = to)
freq(cover) # original frequencies of each land cover class
freq(categ) # note change in frequency of "from" and "to" categories
plot(c(cover, categ))

# Substitute using level name, replace with new level name, replace all others
from <- c("Mosaic crops", "Mosaic cropland/vegetation")
to <- "Crops"
categ <- subst(cover, from = from, to = to, others = "Other")
freq(cover) # original frequencies of each land cover class
freq(categ) # note change in frequency of "from" and "to" categories
plot(c(cover, categ))

}
```

---

sun *Solar radiance and irradiance*

Description

The sun() function calculates beam (direct), diffuse and ground reflected solar irradiation for a given day and set of topographic and atmospheric conditions. The function relies on the **GRASS** module r.sun, the manual page for which contains a detailed explanation (see grassHelp("r.sun"))

Usage

```
sun(
  elevation,
  coeff_bh,
  coeff_dh,
  slope,
  aspect,
  hh,
  horizon_step = 90,
  albedo = 0.2,
  linke = 3,
  day = 1,
  step = 0.5,
  declination = NULL,
  solar_constant = 1367,
  distance_step = 1,
  npartitions = 1,
  beam_rad = TRUE,
  diff_rad = TRUE,
  refl_rad = TRUE,
  glob_rad = TRUE,
  insol_time = TRUE,
  lowMemory = FALSE
)
```

Arguments

elevation	A GRaster with values representing elevation (typically in meters).
coeff_bh	A GRaster: A raster with values of the real-sky beam radiation coefficient. Valid values are between 0 and 1.
coeff_dh	A GRaster: A raster with values of the real-sky diffuse radiation coefficient. Valid values are between 0 and 1.
slope	A GRaster: This is a raster representing topographic slope in radians. It can be generated using terrain().
aspect	A GRaster: This is a raster representing topographic aspect in degrees. It can be generated using terrain(). If generated with that function, "east orientation" *must* be used (i.e., argument northIs0 must be FALSE).
hh	A "stack" of GRasters: This represents height of the horizon in radians in particular directions. Horizon height can be calculated using horizonHeight(). The directions *must* be in "east orientation" (i.e., argument northIs0 in horzionHeight() must be FALSE). The directions must correspond with the sequence given by

horizon_step (see next argument). For example, if horizon_step is 90, then hh must contain rasters representing horizon height at 0 (east), 90 (north), 180 (west), and 270 (south) aspects. #'

horizon_step — Numeric >0: Difference between angular steps in which horizon height is measured. One horizon height raster will be made per value from 0 to 360 - horizon_step degrees.

albedo — A GRaster or a numeric value: This is either a raster with values of ground albedo or a numeric value (in which case albedo is assumed to be the same everywhere). Albedo is unit-less, and the default value is 0.2.

linke — A GRaster or a numeric value: This is either a raster with values of the Linke atmospheric turbidity coefficient or a numeric value (in which case the same value is assumed for all locations). The Linke coefficient is unit-less. The default value is 3, but see also the **GRASS** manual page for module r.sun (grassHelp("r.sun")).

day — Positive integer between 1 to 365, inclusive: Day of year for which to calculate ir/radiation. Default is 1 (January 1st).

step — Positive integer between 0 and 24, inclusive. Time step in hours for all-day radiation sums. Decimal values are OK.

declination — Numeric or NULL (default). Declination value. If NULL, this is calculated automatically.

solar_constant — Positive numeric: The solar constant (solar energy hitting the top of the atmosphere). Default is 1367. Units are W / m^2.

distance_step — Positive numeric between 0.5 and 1.5, inclusive: Sampling distance coefficient. Default is 1.

npartitions — Positive numeric. Number of chunks in which to read input files. Default is 1.

beam_rad — Logical: If TRUE (default), generate a raster with beam irradiation with units of Wh / m^2 / day ("mode 2" of the r.sun **GRASS** module).

diff_rad — Logical: If TRUE (default), generate a raster representing irradiation in Wh / m^2 /day

refl_rad — Logical: If TRUE (default), generate a raster with ground-reflected irradiation with units of Wh / m^2 / day ("mode 2" of the r.sun **GRASS** module).

glob_rad — Logical:. If TRUE (default), generate a raster with total irradiance/irradiation with units of Wh / m^2 / day ("mode 2" of the r.sun **GRASS** module).

insol_time — Logical: If TRUE (default), generate a raster with total insolation time in hours ("mode 2" of the r.sun **GRASS** module).

lowMemory — Logical: If TRUE, use the low-memory version of the r.sun **GRASS** module. The default is FALSE.

Value

A raster or raster stack stack with the same extent, resolution, and coordinate reference system as elevation. Assuming all possible rasters are generated they represent:

- beam_rad: Beam radiation (Watt-hours/m2/day)

- `diff_rad`: Diffuse radiation (Watt-hours/m2/day)
- `refl_rad`: Reflected radiation (Watt-hours/m2/day)
- `glob_rad`: Global radiation (Watt-hours/m2/day)
- `insol_time`: Insolation duration (hours)

See Also

`terrain()`, `horizonHeight()`, **GRASS** manual page for module `r.sun` (see `grassHelp("r.sun")`)

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster
elev <- fast(madElev)

### Calculate input rasters

# Values below are just a guess
coeff_bh <- coeff_dh <- elev
coeff_bh[] <- 0.4
coeff_dh[] <- 0.6

slope <- terrain(elev, "slope")
aspect <- terrain(elev, "aspect", northIs0 = FALSE)

horizon_step <- 90
hh <- horizonHeight(elev, step = horizon_step, northIs0 = FALSE)

### calculate solar ir/radiance

solar <- sun(
 elevation = elev,
 coeff_bh = coeff_bh,
 coeff_dh = coeff_dh,
 slope = slope,
 aspect = aspect,
 hh = hh,
 horizon_step = horizon_step,
 albedo = 0.2,
 linke = 1.5,
 day = 1,
 step = 0.5,
 declination = NULL,
 solar_constant = 1367,
```

```
 distance_step = 1,
 npartitions = 1,

 beam_rad = TRUE,
 diff_rad = TRUE,
 refl_rad = TRUE,
 glob_rad = TRUE,
 insol_time = TRUE,

 lowMemory = FALSE
)

solar

}
```

`terrain,GRaster-method`

Slope, aspect, curvature, and partial slopes

Description

`terrain()` calculates topographic indices, including slope, aspect, curvature, and partial slopes (slopes in the east-west or north-south directions).

Usage

```
## S4 method for signature 'GRaster'
terrain(
  x,
  v = "slope",
  units = "degrees",
  undefinedAspect = NA,
  northIs0 = TRUE
)
```

Arguments

x	A `GRaster` (typically representing elevation).
v	Name of the topographic metric(s) to calculate. Valid values include one or more of: • `"slope"`: Slope. Units are given by argument `units`. • `"aspect"`: Aspect. When argument `northIs0` is `TRUE` (default), then aspect is given in degrees from north going clockwise (0 = north, 90 = east, 180 = south, 270 = west). Units are given by argument `units`. • `"profileCurve"`: Profile curvature.

- "tanCurve": Tangential curvature.
- "dx": Slope in east-west direction.
- "dy": Slope in north-south direction.
- "dxx": Second partial derivative in east-west direction.
- "dyy": Second partial derivative in north-south direction.
- "dxy": Second partial derivative along east-west and north-south direction.
- "*": All of the above.

units — Character: "Units" in which to calculate slope and aspect: either "degrees" for degrees (default), "radians", or "percent". Partial matching is used.

undefinedAspect — Numeric or NA (default): Value to assign to flat areas for which aspect cannot be calculated.

northIs0 — Logical: If TRUE (default), aspect will be reported in "north orientation," such that 0 is north, and degrees run clockwise (90 is east, 180 south, 270 west). If FALSE, then aspect will be reported in "east orientation," such that 0 is east, and degrees run counterclockwise (90 is north, 180 west, 270 south). The latter is the default in **GRASS**, but the former is the default in terra::terrain() function, so is used here as the default. **Note:** The sun() function requires aspect to be in east orientation.

Value

A GRaster with one or more layers.

See Also

terra::terrain(), ruggedness(), wetness(), geomorphons(), module r.slope.aspect in **GRASS**

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster
elev <- fast(madElev)

# Calculate all topographic metrics
topos <- terrain(elev, v = "*")
topos

plot(topos) # NB Aspect has values of NA when it cannot be defined

# Calculate a hillshade raster
```

```
hs <- hillshade(elev)
plot(hs)

}
```

---

thinLines,GRaster-method

Reduce linear features on a raster so linear features are 1 cell wide

---

Description

The thinLines() function attempts to reduce linear features on a raster to just 1 cell wide. You may need to run thinLines() multiple times on the same raster (or experiment with the iter argument) to get acceptable output. thinLines() can be helpful to run on a raster before using as.lines().

Usage

```
## S4 method for signature 'GRaster'
thinLines(x, iter = 200)
```

Arguments

x	A GRaster.
iter	Numeric integer: Number of iterations (default is 200).

Value

A GRaster.

See Also

as.lines(), **GRASS** manual page for module r.thin (see grassHelp("r.thin"))

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Elevation
madElev <- fastData("madElev")

# Convert to GRaster:
elev <- fast(madElev)

# Thin elevation raster:
thinned <- thinLines(elev, iter = 300)
plot(thinned)
```

```
# Convert to lines:
rastToLines <- as.lines(thinned)
plot(rastToLines)

# We can clean this:
cleanLines <- fixDangles(x = rastToLines)
plot(rastToLines, col = "red")
plot(cleanLines, add = TRUE)

}
```

`thinPoints,GVector,GRaster-method`
Reduce number of points in same raster cell

Description

This function thins a "points" `GVector` so that it has no more than `n` points per grid cell in a raster.

Usage

```
## S4 method for signature 'GVector,GRaster'
thinPoints(x, y, n = 1)
```

Arguments

`x`	A "points" `GVector`.
`y`	A `GRaster`.
`n`	Integer or numeric integer: Maximum number of points to remain in a cell. The default is 1.

Value

A "points" `GVector`.

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Elevation and points
madElev <- fastData("madElev")
madDypsis <- fastData("madDypsis")

# Convert to fasterRaster formats:
elev <- fast(madElev)
```

```
dypsis <- fast(madDypsis)

# Aggregate cells of the raster so they are bigger
elevAgg <- aggregate(elev, 32)

# Remove all but one or two points per cell
thin1 <- thinPoints(dypsis, elevAgg, n = 1)
thin2 <- thinPoints(dypsis, elevAgg, n = 2)

# Plot
plot(elevAgg)
plot(dypsis, add = TRUE)
plot(thin2, col = "yellow", add = TRUE)
plot(thin1, col = "red", add = TRUE)
legend(
   "bottomright",
   legend = c("In original & thin 1 & 2",
     "In just thin 1 & 2", "In just thin 1"),
   pch = 16,
   col = c("black", "yellow", "red"),
   bg = "white",
   xpd = NA
)

}
```

`tiles,GRaster-method` *Divide a GRaster into spatially exclusive subsets*

Description

This function divides a `GRaster` into "tiles" or spatial subsets which can be used for speeding some raster calculations. Tiles can be mutually exclusive or overlap by a user-defined number of cells.

Usage

```
## S4 method for signature 'GRaster'
tiles(x, n, overlap = 0, verbose = FALSE)
```

Arguments

`x`	A `GRaster` with one or more layers.
`n`	Numeric vector: Number of tiles to create. This can be a single number, in which case `x` is divided into `n` × `n` tiles, or two values in which case it is divided into `n[1]` × `n[2]` tiles (rows x columns).
`overlap`	Numeric vector (default is 0): Number of rows/columns by which to expand the size of tiles so they overlap. This can be a single value or two values. If just one is provided, the tiles will be expanded by `overlap` rows and `overlaps` columns. If two numbers are provided, the tiles will be expanded by `overlap[1]` rows and `overlap[2]` columns.

verbose — Logical: If TRUE, display progress. Default is FALSE. Progress is only displayed if x is a multi-layer GRaster.

Value

If x has just one layer, then the output is a list with one element per tile. The lapply() and sapply() functions can be used to apply functions to each tile in the list. If x has more than one layer, then the output will be a list of lists, with each sub-list containing the tiles for one GRaster layer.

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Elevation raster
madElev <- fastData("madElev")

# Convert a SpatRaster to a GRaster:
elev <- fast(madElev)

# Create spatially exclusive tiles:
exclusive <- tiles(elev, n = 2, verbose = TRUE)

startpar <- par(mfrow = c(2, 3))
plot(elev, main = "Original")

for (i in seq_along(exclusive)) {
 plot(exclusive[[i]], ext = elev, main = paste("Tile", i))
}
par(startpar)

# Create tiles that overlap:
overlaps <- tiles(elev, n = 2, overlap = 200, verbose = TRUE)

startpar <- par(mfrow = c(2, 3))
plot(elev, main = "Original")

for (i in seq_along(overlaps)) {
 plot(overlaps[[i]], ext = elev, main = paste("Tile", i))
}
par(startpar)

}
```

---

topology,GSpatial-method

Topology (2- or 3-dimensions) of a GRaster or GVector

Description

GRasters and GVectors can have 2-dimensional or 3-dimensional coordinates. This function returns the dimensions of the object.

Usage

```
## S4 method for signature 'GSpatial'
topology(x)
```

Arguments

x	A GSpatial object (i.e., a GRaster or GVector).

Value

Either "2D" or "3D".

See Also

is.2d(), is.3d()

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")
madForest2000 <- fastData("madForest2000")

# Convert SpatRasters to GRasters
elev <- fast(madElev)
forest <- fast(madForest2000)

### GRaster properties

# plotting
plot(elev)

# dimensions
dim(elev) # rows, columns, depths, layers
nrow(elev) # rows
ncol(elev) # columns
ndepth(elev) # depths
nlyr(elev) # layers

res(elev) # resolution (2D)
res3d(elev) # resolution (3D)
zres(elev) # vertical resolution
```

```
xres(elev) # vertical resolution
yres(elev) # vertical resolution
zres(elev) # vertical resolution (NA because this is a 2D GRaster)

# cell counts
ncell(elev) # cells
ncell3d(elev) # cells (3D rasters only)

# number of NA and non-NA cells
nacell(elev)
nonnacell(elev)

# topology
topology(elev) # number of dimensions
is.2d(elev) # is it 2-dimensional?
is.3d(elev) # is it 3-dimensional?

minmax(elev) # min/max values

# "names" of the object
names(elev)

# coordinate reference system
crs(elev)
st_crs(elev)
coordRef(elev)

# extent (bounding box)
ext(elev)

# vertical extent (not defined for this raster)
zext(elev)

# data type
datatype(elev) # fasterRaster type
datatype(elev, "GRASS") # GRASS type
datatype(elev, "terra") # terra type
datatype(elev, "GDAL") # GDAL type

is.integer(elev)
is.float(elev)
is.double(elev)
is.factor(elev)

# convert data type
as.int(elev) # integer; note that "elev" is already of type "integer"
as.float(elev) # floating-precision
as.doub(elev) # double-precision

# assigning
pie <- elev
pie[] <- pi # assign all cells to the value of pi
pie
```

```
# concatenating multiple GRasters
rasts <- c(elev, forest)
rasts

# subsetting
rasts[[1]]
rasts[["madForest2000"]]

# replacing
rasts[[2]] <- 2 * forest
rasts

# adding layers
rasts[[3]] <- elev > 500 # add a layer
rasts <- c(rasts, sqrt(elev)) # add another
add(rasts) <- ln(elev)
rasts

# names
names(rasts)
names(rasts) <- c("elev_meters", "2_x_forest", "high_elevation", "sqrt_elev", "ln_elev")
rasts

# remove a layer
rasts[["2_x_forest"]] <- NULL
rasts

# number of layers
nlyr(rasts)

# correlation and covariance matrices
madLANDSAT <- fastData("madLANDSAT")
landsat <- fast(madLANDSAT) # projects matrix
layerCor(landsat) # correlation
layerCor(landsat, fun = 'cov') # covariance

}
```

---

trim,GRaster-method *Remove rows and columns from a raster that are all NA*

Description

This function removes any rows and columns from a GRaster that are all NA.

If the GRaster is a stack of rasters, then the rasters will all be trimmed to the same extent such that none have any rows or columns that are all NA. In other words, if at least one raster in the stack has a non-NA cell in a row or column, all rasters will retain that row or column.

Usage

```
## S4 method for signature 'GRaster'
trim(x, pad = 0)
```

Arguments

x	A GRaster.
pad	Numeric integer: Number of NA rows and columns to retain. The default is 0.

Value

A GRaster.

See Also

terra::trim(), extend(), and **GRASS** module g.region

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Elevation raster
madElev <- fastData("madElev")

# Convert SpatRaster to a GRaster:
elev <- fast(madElev)

# Trim NA rows/columns:
trimmedElev <- trim(elev)
dim(elev)
dim(trimmedElev)

# Trim a "stack" of rasters. We will artificially add NA rows and columns to
# one raster to demonstrate how the trim() function removes only rows/columns
# that are NA for *all* rasters.
elevNAs <- elev

fun <- " = if(col() > ncols() - 200, null(), madElev)"
elevNAs <- app(elevNAs, fun)

# Notice raster is "narrower" because we added NA columns
plot(elevNAs)

elevs <- c(elev, elevNAs)
trimmedElevs <- trim(elevs)
trimmedElevNAs <- trim(elevNAs)

dim(elevs)
```

```
dim(trimmedElevNAs)
dim(trimmedElevs)

}
```

---

union,GVector,GVector-method
Combine two GVectors

Description

The union() function combines two "polygons" GVectors. The output will have at least as many geometries as both GVectors, plus more if polygons of one divide polygons of the other, and vice versa. You can also use the + operator (e.g., vect1 + vect2).

Usage

```
## S4 method for signature 'GVector,GVector'
union(x, y)
```

Arguments

x, y	GVectors representing polygons.

Value

A GVector.

See Also

c(), aggregate(), crop(), intersect(), xor(), erase()

Examples

```
if (grassStarted()) {

# Setup
library(sf)

# Polygon of coastal Madagascar and Dypsis specimens
madCoast4 <- fastData("madCoast4") # polygons
madDypsis <- fastData("madDypsis") # points

# Convert vectors:
coast4 <- fast(madCoast4)
dypsis <- fast(madDypsis)

# Create another polygons vector from a convex hull around Dypsis points
hull <- convHull(dypsis)
```

```
### union()

unioned <- union(coast4, hull)
plot(unioned)

plus <- coast4 + hull # same as union()

### intersect

inter <- intersect(coast4, hull)
plot(coast4)
plot(hull, border = "red", add = TRUE)
plot(inter, border = "blue", add = TRUE)

### xor

xr <- xor(coast4, hull)
plot(coast4)
plot(xr, border = "blue", add = TRUE)

### erase

erased <- erase(coast4, hull)
plot(coast4)
plot(erased, border = "blue", add = TRUE)

minus <- coast4 - hull # same as erase()

}
```

`update,GRaster-method` *Refresh metadata in a GRaster or GVector*

Description

GRasters and GVectors are really pointers to objects in **GRASS**. The values displayed when you use `show()` or `print()` for a `GRaster` or `GVector` are stored in **R**. If, on the odd chance that you make a change to a `GRaster` or `GVector` (e.g., using commands in the **rgrass** package), the changes will not be automatically reflected in the `GRaster` or `GVector`. This function can be used to update the objects in **R** to reflect their proper values.

Usage

```
## S4 method for signature 'GRaster'
update(object)

## S4 method for signature 'GVector'
update(object)
```

Arguments

object	A GRaster or GVector.

Value

A GRaster or GVector.

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")
madForest2000 <- fastData("madForest2000")

# Convert SpatRasters to GRasters
elev <- fast(madElev)
forest <- fast(madForest2000)

### GRaster properties

# plotting
plot(elev)

# dimensions
dim(elev) # rows, columns, depths, layers
nrow(elev) # rows
ncol(elev) # columns
ndepth(elev) # depths
nlyr(elev) # layers

res(elev) # resolution (2D)
res3d(elev) # resolution (3D)
zres(elev) # vertical resolution
xres(elev) # vertical resolution
yres(elev) # vertical resolution
zres(elev) # vertical resolution (NA because this is a 2D GRaster)

# cell counts
ncell(elev) # cells
ncell3d(elev) # cells (3D rasters only)

# number of NA and non-NA cells
nacell(elev)
nonnacell(elev)

# topology
topology(elev) # number of dimensions
is.2d(elev) # is it 2-dimensional?
```

```
is.3d(elev) # is it 3-dimensional?

minmax(elev) # min/max values

# "names" of the object
names(elev)

# coordinate reference system
crs(elev)
st_crs(elev)
coordRef(elev)

# extent (bounding box)
ext(elev)

# vertical extent (not defined for this raster)
zext(elev)

# data type
datatype(elev) # fasterRaster type
datatype(elev, "GRASS") # GRASS type
datatype(elev, "terra") # terra type
datatype(elev, "GDAL") # GDAL type

is.integer(elev)
is.float(elev)
is.double(elev)
is.factor(elev)

# convert data type
as.int(elev) # integer; note that "elev" is already of type "integer"
as.float(elev) # floating-precision
as.doub(elev) # double-precision

# assigning
pie <- elev
pie[] <- pi # assign all cells to the value of pi
pie

# concatenating multiple GRasters
rasts <- c(elev, forest)
rasts

# subsetting
rasts[[1]]
rasts[["madForest2000"]]

# replacing
rasts[[2]] <- 2 * forest
rasts

# adding layers
rasts[[3]] <- elev > 500 # add a layer
```

```
rasts <- c(rasts, sqrt(elev)) # add another
add(rasts) <- ln(elev)
rasts

# names
names(rasts)
names(rasts) <- c("elev_meters", "2_x_forest", "high_elevation", "sqrt_elev", "ln_elev")
rasts

# remove a layer
rasts[["2_x_forest"]] <- NULL
rasts

# number of layers
nlyr(rasts)

# correlation and covariance matrices
madLANDSAT <- fastData("madLANDSAT")
landsat <- fast(madLANDSAT) # projects matrix
layerCor(landsat) # correlation
layerCor(landsat, fun = 'cov') # covariance

}
```

`vect,GVector-method` *Convert a GVector to a SpatVector or sf vector*

Description

The **fasterRaster** version of the `vect()` function converts a `GVector` to a `SpatVector` (from the **terra** package). The **fasterRaster** version of the `st_as_sf()` function converts a `GVector` to an `sf` object (**sf** package).

Usage

```
## S4 method for signature 'GVector'
vect(x, ...)

## S4 method for signature 'GVector'
st_as_sf(x)
```

Arguments

`x`	A `GVector`.
`...`	Additional arguments to send to `writeVector()`.

Value

`vect()` returns a `SpatVector` (**terra** package), and `st_as_sf()` returns an `sf` vector (**sf** package).

See Also

terra::vect(), sf::st_as_sf()

Examples

```
if (grassStarted()) {

# Setup
library(sf)

# Example data:
madCoast4 <- fastData("madCoast4")
madRivers <- fastData("madRivers")
madDypsis <- fastData("madDypsis")

# Convert sf vectors to GVectors:
coast <- fast(madCoast4)
rivers <- fast(madRivers)
dypsis <- fast(madDypsis)

# Geographic properties:
ext(rivers) # extent
crs(rivers) # coordinate reference system
st_crs(rivers) # coordinate reference system
coordRef(rivers) # coordinate reference system

# Column names and data types:
names(coast)
datatype(coast)

# Points, lines, or polygons?
geomtype(dypsis)
geomtype(rivers)
geomtype(coast)

is.points(dypsis)
is.points(coast)

is.lines(rivers)
is.lines(dypsis)

is.polygons(coast)
is.polygons(dypsis)

# Number of dimensions:
topology(rivers)
is.2d(rivers) # 2-dimensional?
is.3d(rivers) # 3-dimensional?

# Just the data table:
as.data.frame(rivers)
as.data.table(rivers)
```

```
# Top/bottom of the data table:
head(rivers)
tail(rivers)

# Vector or table with just selected columns:
names(rivers)
rivers$NAME
rivers[[c("NAM", "NAME_0")]]
rivers[[c(3, 5)]]

# Select geometries/rows of the vector:
nrow(rivers)
selected <- rivers[2:6]
nrow(selected)

# Plot:
plot(coast)
plot(rivers, col = "blue", add = TRUE)
plot(selected, col = "red", lwd = 2, add = TRUE)

# Vector math:
hull <- convHull(dypsis)

un <- union(coast, hull)
sameAsUnion <- coast + hull
plot(un)
plot(sameAsUnion)

inter <- intersect(coast, hull)
sameAsIntersect <- coast * hull
plot(inter)
plot(sameAsIntersect)

er <- erase(coast, hull)
sameAsErase <- coast - hull
plot(er)
plot(sameAsErase)

xr <- xor(coast, hull)
sameAsXor <- coast / hull
plot(xr)
plot(sameAsXor)

# Vector area and length:
expanse(coast, unit = "km") # polygons areas
expanse(rivers, unit = "km") # river lengths

### Fill holes

# First, we will make some holes by creating buffers around points.
buffs <- buffer(dypsis, 500)
```

```
holes <- coast - buffs
plot(holes)

filled <- fillHoles(holes, fail = FALSE)

}
```

`vegIndex,GRaster-method`

Vegetation indices from surface reflectance

Description

This function calculates one of many types of vegetation indices from a raster with four bands representing blue (B), green (G), red (R), and near infrared (NIR), plus possibly channels 5 and 7. The function requires rasters that represent surface reflectance, so should have values that fall in the range 0 to 1, unless they are digital number rasters (e.g., integers in the range 0 to 255). If digital number format is used, then the `bits` argument should be defined.

Usage

```
## S4 method for signature 'GRaster'
vegIndex(
  x,
  index = "NDVI",
  r = NULL,
  g = NULL,
  b = NULL,
  nir = NULL,
  b5 = NULL,
  b7 = NULL,
  soilSlope = NULL,
  soilIntercept = NULL,
  soilNR = 0.08,
  bits = NULL
)
```

Arguments

`x`	A `GRaster` with one layer per required band. Values should be between 0 and 1.
`index`	Character or character vector: The vegetation index or indices to calculate. You can find a list of available indices using fastData("vegIndices") (also see vegIndices). The first column, "`index`" provides the name of the index, and these are the values that this argument will accept (e.g., "NDVI", "EVI2"). Partial matching is used, and case is ignored. You can also use these shortcuts: • `"*"`: Calculate *all* indices

- "RNIR": Calculate all indices that use R and NIR channels (but not other channels).
- "NotSoil": Calculate all indices that use any channels but do not require soilSlope or soilIntercept.

Note: A near-comprehensive table of indices can be found on the Index Database: A Database for Remote Sensing Indices.

r, g, b, nir
: Numeric or character: Index or names() of the layers in x that represent the red, green, blue, and near infrared channels. Values must be in the range from 0 to 1 or integers.

b5, b7
: Numeric or character: Index of names of the layers representing bands 5 and 7. These are used only for GVI and PVI. Values must be in the range from 0 to 1 or integers.

soilSlope, soilIntercept, soilNR
: Numeric: Values of the soil slope, intercept, and soil noise reduction factor (0.08, by default). Used only for calculation of MSAVI.

bits
: Either NULL (default) or numeric integer or integer with a value of 7, 8, 10, or 16: If the rasters are represented by integers (so do not fall in the range of 0 to 1), then the number of bits can be supplied using bits. If this is the case, then they will range from 0 to 2^n, where n is 7, 8, 10, or 16. If bit rasters are supplied, they must be of datatype() "integer". If raster values are in the range from 0 to 1, then bits should be NULL (default).

Value

A GRaster.

See Also

GRASS manual page for module i.vi (see grassHelp("i.vi"))

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Elevation raster, rivers vector
madLANDSAT <- fastData("madLANDSAT")

# Convert a SpatRaster to a GRaster:
landsat <- fast(madLANDSAT)

# See available vegetation indices:
vegIndices

# Normalized Difference Vegetation Index and Enhanced Vegetation Index:
indices <- c("ndvi", "evi")
vi <- vegIndex(landsat, index = indices, r = 1, b = 3, nir = 4, bits = 8)
```

```
plot(vi)

# All indices using R and NIR:
rnir <- vegIndex(landsat, index = "rnir", r = 1, nir = 4, bits = 8)

# Note: Some values are highly skewed
plot(rnir)

}
```

---

vegIndices — *Table of vegetation indices that can be calculated from remote sensing surface reflectance data using* vegIndex(). *A near-comprehensive table of indices can be found on the* R*hrefhttps://www.indexdatabase.deIndex Database: A Database for Remote Sensing Indices.*

Description

A table of vegetation indices that ca be calculated using vegIndex(). Columns include:

- 'index": Abbreviation of the index.
- definition: Index name
- R, G, B, NIR, channel5, channel7: Whether or not the index uses the red, green, blue, or near-infrared channels, and channels 5 and 7.
- soilLineslope, soilIntercept, soilNR: Whether or not the index requires soil line slope, soil intercept, and a soil noise reduction factor.

Format

An object of class data.frame.

See Also

vegIndex()

Examples

```
### vector data

library(sf)

# For vector data, we can use data(*) or fastData(*):
data(madCoast0) # same as next line
madCoast0 <- fastData("madCoast0") # same as previous
madCoast0
plot(st_geometry(madCoast0))
```

```
madCoast4 <- fastData("madCoast4")
madCoast4
plot(st_geometry(madCoast4), add = TRUE)

madRivers <- fastData("madRivers")
madRivers
plot(st_geometry(madRivers), col = "blue", add = TRUE)

madDypsis <- fastData("madDypsis")
madDypsis
plot(st_geometry(madDypsis), col = "red", add = TRUE)

### raster data

library(terra)

# For raster data, we can get the file directly or using fastData(*):
rastFile <- system.file("extdata/madElev.tif", package="fasterRaster")
madElev <- terra::rast(rastFile)

madElev <- fastData("madElev") # same as previous two lines
madElev
plot(madElev)

madForest2000 <- fastData("madForest2000")
madForest2000
plot(madForest2000)

madForest2014 <- fastData("madForest2014")
madForest2014
plot(madForest2014)

# multi-layer rasters
madChelsa <- fastData("madChelsa")
madChelsa
plot(madChelsa)

madPpt <- fastData("madPpt")
madTmin <- fastData("madTmin")
madTmax <- fastData("madTmax")
madPpt
madTmin
madTmax


# RGB raster
madLANDSAT <- fastData("madLANDSAT")
madLANDSAT
plotRGB(madLANDSAT, 4, 1, 2, stretch = "lin")

# categorical raster
madCover <- fastData("madCover")
madCover
```

```
madCover <- droplevels(madCover)
levels(madCover) # levels in the raster
nlevels(madCover) # number of categories
catNames(madCover) # names of categories table

plot(madCover)
```

`voronoi,GVector-method`
Voronoi tessellation

Description

This function creates a Voronoi tessellation from a set of spatial points or polygons.

Usage

```
## S4 method for signature 'GVector'
voronoi(x, buffer = 0)
```

Arguments

`x`	A `GVector` "points" object.
`buffer`	Numeric: By default, this function creates a vector that has an extent exactly the same as the input data. However, the apparent extent can be changed by setting this value to a value different from 0. Negative values reduce the size of the extent, and positive extend it. Units are in map units.

Value

A `GVector`.

See Also

`terra::voronoi()`, `sf::st_voronoi()`, module `v.voronoi` in **GRASS**

Examples

```
if (grassStarted()) {

# Setup
library(sf)

# Example vectors
madDypsis <- fastData("madDypsis") # points
madCoast4 <- fastData("madCoast4") # polygons

# Convert sf vectors to GVectors
dypsis <- fast(madDypsis)
```

```
coast4 <- fast(madCoast4)
ant <- coast4[coast4$NAME_4 == "Antanambe"]

# Delaunay triangulation
dypsisDel <- delaunay(dypsis)
plot(dypsisDel)
plot(dypsis, pch = 1, col = "red", add = TRUE)

# Voronoi tessellation
vor <- voronoi(dypsis)
plot(vor)
plot(dypsis, pch = 1, col = "red", add = TRUE)

# Random Voronoi tessellation
rand <- rvoronoi(coast4, size = 100)
plot(rand)

}
```

`wetness,GRaster-method`

Topographic wetness index

Description

This function creates a raster map with values equal to the topographic wetness index (TWI), which is a measure of how much overland water flow tends to accumulate in or flow away from a location.

Usage

```
## S4 method for signature 'GRaster'
wetness(x)
```

Arguments

`x`	A `GRaster` (typically representing elevation). The raster must be projected (i.e., not in WGS84, NAD83, et cetera).

Value

A `GRaster`.

See Also

`terrain()`, `ruggedness()`, `geomorphons()`, **GRASS** manual for module `r.topidx` (see `grassHelp("r.topidx")`)

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Elevation raster
madElev <- fastData("madElev")

# Convert to GRaster:
elev <- fast(madElev)

# Terrain ruggedness index:
tri <- ruggedness(elev)
plot(c(elev, tri))

# Topographic wetness index:
twi <- wetness(elev)
plot(c(elev, twi))

}
```

`writeRaster,GRaster,character-method`
Save a GRaster to disk

Description

This function saves a `GRaster` to disk directly from a **GRASS** session. It is faster than using `rast()`, then saving the output of that to disk (because `rast()` actually save the raster to disk, anyway).

The function will attempt to ascertain the file type to be ascertained from the file extension, but you can specify the format using the `format` argument (see entry for `...`). You can see a list of supported formats by simply using this function with no arguments, as in `writeRaster()`, or by consulting the online help page for the **GRASS** module `r.out.gdal` (see `grassHelp("r.out.gdal")`). Only the `GeoTIFF` file format is guaranteed to work for multi-layered rasters.

The function will attempt to optimize the `datatype` argument, but this can take a long time. You can speed this up by setting `datatype` manually. Note that if you are saving a "stack" of `GRasters` with different `datatypes`, the one with the highest information density will be used (e.g., low-bit integer < high-bit integer < floating-point < double-floating point). This can make rasters with lower datatypes much larger on disk. In these cases, it make be best to save rasters with similar `datatypes` together.

Usage

```
## S4 method for signature 'GRaster,character'
writeRaster(
  x,
  filename,
```

```
  overwrite = FALSE,
  datatype = NULL,
  byLayer = FALSE,
  names = TRUE,
  levelsExt = NULL,
  compress = "LZW",
  warn = TRUE,
  ...
)

## S4 method for signature 'missing,missing'
writeRaster(x, filename)
```

Arguments

x	A GRaster or missing: If missing, a table of supported file types is reported.
filename	Character: Path and file name.
overwrite	Logical: If FALSE (default), do not save over existing file(s).
datatype	NULL (default) or character: The datatype of the values stored in non-ASCII rasters. If NULL, this will be ascertained from the raster, and the function usually does a good job at it. However, you can force it manually, but note that in some cases, trying to save a GRaster using an inappropriate datatype for its values can result in an error or in the function exiting without an error but also without having written the raster to disk. The argument can take any of those shown below under the first four columns, but whatever is used, it will be converted to the **GDAL** version.

fasterRaster	terra	GRASS	GDAL	Values
integer	INT1U	CELL	Byte	Integer values from 0 to 255
integer	INT2U	CELL	UInt16	Integer values from 0 to 65,534
integer	INT2S	CELL	Int16	Integer values from -32,767 to -32,767
integer	INT4S	CELL	Int32	Integer values from -2,147,483,647 to 2,147,483,647
float	FLT4S	FCELL	Float32	Values from -3.4E+38 to 3.4E+38, including decimal values
double	FLT8S	DCELL	Float64	Values from -1.79E+308 to 1.79E+308, including decimal values
factor	INT*	CELL	INT*	Integer values corresponding to categories

* Depends on the integers (signed/unsigned, range of values). Categorical rasters will have an associated file saved with them that has category values and labels. The file name will be the same as the raster's file name, but end with the extension given by levelsExt (.csv by default).

byLayer	Logical: If FALSE (default), multi-layer rasters will be saved in one file. If TRUE, the each layer will be saved in a separate file. The filename from filename will be amended so that it ends with _<name> (then the file extension), where <name> is give by names(). Note that if any characters in raster names will not work in a file name, then the function will fail (e.g., a backslash or question mark).

names — Logical: If TRUE (default), save a file with raster layer names. The file will have the same name as the raster file but end with "_names.csv". Currently, the names() attribute of rasters cannot be saved in the raster, which can create confusion when multi-layered rasters are saved. Turning on this option will save the ancillary file with layer names. If it exists, this file will be read by fast() so layer names are assigned when the raster is read by that function. The absence of a "names" file will not create any issues with this function or fast(), other than not having the metadata on layer names.

levelsExt — Character, logical, or NULL (default): Name of the file extension for the "levels" file that accompanies a categorical GRaster. When saving categorical rasters, the raster file is accompanied with a "levels" file that contain information on the levels of the raster. This file is the same as filename, except it has a different extension. Valid values depend on how many raster layers are saved at a time (case is ignored):

- DefaultOne raster layer: ".csv"
- Two or more layers, with at least one categorical raster: ".rds", ".rda", ".rdat", ".rdata"
- Any: NULL or TRUE automatically selects either ".csv" (one raster layer) or ".rds (two or more)
- Any: FALSE disables saving of a levels file.

compress — Character: Type of compression to use for GeoTIFF files:

- "LZW" (default)
- "DEFLATE"
- "PACKBITS"
- "LZMA"
- NULL: No compression is used, but the file can still be reduced in size by using zip, gzip, or other compressions.

warn — Logical: If TRUE (default), display a warning if the datatype argument does not match the value given by datatype(x, "GDAL"), or if the fileExt argument will not work with the given raster and so has been automatically changed.

... — Additional arguments. These can include:

- bigTiff: Logical: If TRUE, and the file format is a GeoTIFF and would be larger than 4 GB (regardless of compression), then the file will be saved in BIGTIFF format.
- format: Character, indicating file format. This is usually ascertained from the file extension, but in case this fails, it can be stated explicitly. When using other formats, you may have to specify the createopts argument, too (see help page for **GRASS** module r.out.gdal). Two common formats include:
  - "GTiff" (default): GeoTIFF filename ends in .tif.
  - "ASC": ASCII filename ends in .asc
- Additional arguments to send to **GRASS** modules r.out.gdal and r.out.ascii.
- precision: Numeric: For ASCII files, you may need to state the number of significant digits. 32-bit values have 7 digits and 64-bit values have 16. So in these cases the argument would be precision=7 or precision=16.

Value

A GRaster (invisibly). A raster is also saved to disk.

See Also

terra::writeRaster(), **GRASS** module r.out.gdal (see grassHelp("r.out.gdal"))

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madElev <- fastData("madElev")
madChelsa <- fastData("madChelsa")

### What raster formats can we attempt to write?
writeRaster()

### Save GRaster to disk (using temporary file)
elev <- fast(madElev)
filename <- tempfile(fileext = ".tif")
writeRaster(elev, filename)

# Load raster from disk
elev2 <- fast(filename)
elev2

### Save multi-layer GRaster to disk in one file (using temporary file)
chelsa <- fast(madChelsa)
filename <- tempfile(fileext = ".tif")
writeRaster(chelsa, filename)

# Load raster from disk
chelsa2 <- fast(filename)
chelsa2

### Save multi-layer GRaster to disk layer-by-layer (using temporary file)
chelsa <- fast(madChelsa)
filename <- tempfile(fileext = ".tif")
writeRaster(chelsa, filename, byLayer = TRUE)

# Load one of the rasters from disk
filename2 <- sub(filename, pattern = ".tif", replacement = "_bio1.tif")
chelsaBio1 <- fast(filename2)
chelsaBio1

}
```

writeVector,GVector,character-method
Save a GVector to disk

Description

This function saves a GVector to disk directly from a **GRASS** session.

By default, files will be of OGC GeoPackage format (extension ".gpkg"), but this can be changed with the format argument. You can see a list of supported formats by simply using this function with no arguments, as in writeVector(), or by consulting the online help page for **GRASS** module v.out.ogr (see grassHelp("v.out.ogr")).

Note that if the vector has a data table attached and at least one numeric or integer column has an NA or NaN value, the function will yield a warning like:

```
Warning 1: Invalid value type found in record 2 for field column_with_NA_or_NaN. This warning will no lor
```

Also note that despite the promise, this warning will be displayed again.

Usage

```
## S4 method for signature 'GVector,character'
writeVector(
  x,
  filename,
  overwrite = FALSE,
  format = NULL,
  attachTable = TRUE,
  ...
)

## S4 method for signature 'missing,missing'
writeVector(x, filename)
```

Arguments

x	A GVector.
filename	Character: Path and file name.
overwrite	Logical: If FALSE (default), do not save over existing files.
format	Character or NULL: File format. If NULL (default), then the function will attempt to get the format from the file name extension. Partial matching is used and case is ignored. You can see a list of formats using writeVector() (no arguments). Some common formats include: • "GPKG": OGC GeoPackage (extension .gpkg). • "CSV": Comma-separated value... saves the data table only, not the geometries (extension .csv).

- "ESRI Shapefile": ESRI shapefile (extension .shp).
- "GeoJSON": GeoJSON (extension GeoJSON)
- "KML": Keyhole Markup Language (extension .kml)
- "netCDF": NetCDF (extension .ncdf)
- "XLSX": MS Office Open XML spreadsheet (extension .xlsx).

attachTable — Logical: If TRUE (default), attach the attribute to table to the vector before saving it. If FALSE, the attribute table will not be attached.

... — Additional arguments to send to **GRASS** module v.out.ogr (see grassHelp("v.out.ogr")).

Value

Invisibly returns a GRaster (the input, x). Also saves the vector to disk.

See Also

terra::writeVector(), sf::st_write(), **GRASS** module v.out.ogr (see grassHelp("v.out.ogr"))

terra::writeVector(), the **GRASS** module manual page for v.out.ogr (see grassHelp("v.out.ogr"))

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data
madRivers <- fastData("madRivers")

# What file formats can we attempt to write?
writeVector()

# Convert SpatVector to GVector
rivers <- fast(madRivers)
rivers

# Save GVector to disk as GeoPackage
filename <- tempfile(fileext = ".gpkg")
writeVector(rivers, filename)

# Save GVector to disk as ESRI Shapefile
filename <- tempfile(fileext = ".shp")
writeVector(rivers, filename)

# Save GVector to disk as Google Earth KML
filename <- tempfile(fileext = ".klm")
writeVector(rivers, filename)

# Save GVector data table to disk as comma-separated file
filename <- tempfile(fileext = ".csv")
writeVector(rivers, filename)
```

```
# Save GVector data table to disk as NetCDF
filename <- tempfile(fileext = ".ncdf")
writeVector(rivers, filename)

# Save GVector data table to disk as Excel file
filename <- tempfile(fileext = ".xlsx")
writeVector(rivers, filename)

}
```

---

xor,GVector,GVector-method

Select parts of polygons not shared between two GVectors

Description

The xor() function selects the area that does *not* overlap between two "polygon" GVectors. You can also use the / operator, as in vect1 / vect2.

Usage

```
## S4 method for signature 'GVector,GVector'
xor(x, y)
```

Arguments

x, y	GVectors.

Value

A GVector.

See Also

crop(), intersect(), union(), erase()

Examples

```
if (grassStarted()) {

# Setup
library(sf)

# Polygon of coastal Madagascar and Dypsis specimens
madCoast4 <- fastData("madCoast4") # polygons
madDypsis <- fastData("madDypsis") # points

# Convert vectors:
```

```
coast4 <- fast(madCoast4)
dypsis <- fast(madDypsis)

# Create another polygons vector from a convex hull around Dypsis points
hull <- convHull(dypsis)

### union()

unioned <- union(coast4, hull)
plot(unioned)

plus <- coast4 + hull # same as union()

### intersect

inter <- intersect(coast4, hull)
plot(coast4)
plot(hull, border = "red", add = TRUE)
plot(inter, border = "blue", add = TRUE)

### xor

xr <- xor(coast4, hull)
plot(coast4)
plot(xr, border = "blue", add = TRUE)

### erase

erased <- erase(coast4, hull)
plot(coast4)
plot(erased, border = "blue", add = TRUE)

minus <- coast4 - hull # same as erase()

}
```

`zonal,GRaster,ANY-method`

Statistics on cells of a GRaster stratified by cells of another raster

---

Description

Function `zonal()` calculates statistics (mean, sum, etc.) on cells of a `GRaster` by "zones" created by cells of another `GRaster` or `GVector`.

Usage

```
## S4 method for signature 'GRaster,ANY'
zonal(x, z, fun = "mean", probs = 0.5)
```

Arguments

x	A GRaster for which to calculate summary statistics.
z	A GRaster or GVector used to define zones: • If z is a GRaster, then it must be of type integer or factor (see vignette("GRasters", package = "fasterRaster")). Zones will be established based on cells that have the same value in this raster. • If z is a GVector, zones will be created for each geometry. If geometries overlap, then the zonal statistics will be calculated for the ones on top. Thus statistics for the zones defined by geometries below these may not represent all the cells covered by that geometry.
fun	Character vector: Name of the function(s) to summarize x with. These can include: • "*": All of the functions below. • "cv": Sample coefficient of variation (expressed as a proportion of the mean). • "cvpop": Population coefficient of variation (expressed as a proportion of the mean). • "max" and "min": Highest and lowest values across non-NA cells. • "mean" (default): Average. • "meanAbs": Mean of absolute values. • "median": Median. • "quantile": Quantile (see also argument probs). • "range": Range. • "sd": Sample standard deviation. • "sdpop": Population standard deviation. • "sum": Sum. • "var": Sample variance. • "varpop": Population variance.
probs	Numeric: Quantile at which to calculate quantile. Only a single value between 0 and 1 is allowed.

Value

A data.frame or data.table.

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Elevation SpatRaster:
madElev <- fastData("madElev")
```

```
# Convert a SpatRaster to a GRaster:
elev <- fast(madElev)

### Calculate zonal statistics using a GRaster as zones

# Generate a "zones" GRaster by dividing raster into areas based on
# high/low elevation.
names(elev) # Use this name in app() formula.
fun <- "= if (madElev <200, 0, if (madElev <400, 1, 2))"
zones <- app(elev, fun = fun)

# Calculate zonal statistics using a raster as zones
zonal(elev, zones, fun = "mean")
zonal(elev, zones, fun = "*") # all statistics

# Calculate zonal statistics on multi-layered GRaster
elev2 <- c(elev, log10(elev))
zonal(elev2, zones, fun = c("mean", "sum", "sdpop"))

### Calculate zonal statistics using a GVector as zones

madCoast4 <- fastData("madCoast4")
coast <- fast(madCoast4)

zonal(elev, z = coast, fun = "mean")

}
```

`zonalGeog,GRaster-method`

Geographic statistics for sets of cells with the same values

Description

This function calculates geographic statistics for each set of cells in an `integer` or `factor` `GRaster`. Statistics include:

- Area
- Perimeter length
- "Compact square" statistic: $4\sqrt{(area)}/perimeter)$
- "Compact circle" statistic: $4 * P/(2\sqrt{(\pi * A)})$ where *P* is the perimeter length and *A* the area.
- fractal dimension: $2(log(P)/log(A + 0.001))$ where *P* is perimeter length and *A* is area.
- The average x- and y-coordinates of each zone.

Usage

```
## S4 method for signature 'GRaster'
zonalGeog(x, unit = "meters")
```

Arguments

x	A GRaster.
unit	Character: Units of the output. Any of: • "meters" (default) • "kilometers" or "km" • "miles" or "mi" • "yards" or "yd" • "feet" or "ft": International foot; 1 foot exactly equal to 0.3048 meters • "cells": Number or cells Partial matching is used and case is ignored.

Value

A list of data.frames or a data.tables, one per layer in x. Only layers that are integers or factors have their geographies calculated. Other layers have NULL tables returned.

Examples

```
if (grassStarted()) {

# Setup
library(terra)

# Example data: Elevation and land cover
madElev <- fastData("madElev")
madForest2000 <- fastData("madForest2000")
madCover <- fastData("madCover")

# Convert to GRasters:
elev <- fast(madElev)
forest2000 <- fast(madForest2000)
cover <- fast(madCover)

# Rename
names(elev) <- "elev"
names(forest2000) <- "forest"

# Geometric statistics for an integer raster zoned by elevation:
fun <-
  "= if (elev <400 & forest == 1, 0, if (elev >=400 & forest == 1, 1, null()))"
forestByElev <- app(c(elev, forest2000), fun = fun)
plot(forestByElev, main = "forest < 400 m & >= 400 m")
zonalGeog(forestByElev)

# Geometric statistics for a categorical raster:
zonalGeog(cover)

}
```

[*Subset geometries of a GVector*

Description

The [operator returns a subset or remove specific geometries of a GVector. You can get the number of geometries using ngeom(). Note that you cannot use this function to change the "order" in which geometries or their associated records in a data table appear. For example, vector[1:3] and vector[3:1] will yield the exact same results.

Note that subsetting can take a very long time if you are retaining only a small number of geometries from a vector with many geometries. The routine selects geometries by removing those that are not in i. So if you can write code to remove fewer geometries (i.e., an "inverse" selection), it may go faster.

Usage

```
## S4 method for signature 'GVector,ANY,ANY'
x[i, j]

## S4 method for signature 'GRaster,GRaster,ANY'
x[i, j]
```

Arguments

x	A GVector.
i	Numeric integer, integer, or logical vector: Indicates which geometry(ies) to obtain. Negative numeric or integer values will remove the given geometries from the output. If a logical vector is supplied and it is not the same length as the number of geometries, it will be recycled.
j	Numeric integer, integer, logical, or character: Indices or name(s) of the column(s) to obtain. You can see column names using names(). Negative numeric or integer values will remove the given columns from the output. If a logical vector is supplied and it is not the same length as the number of columns, it will be recycled.

Value

A GVector.

See Also

subset(), $, [[

Examples

```
if (grassStarted()) {

# Setup
library(terra)

### GRasters

# Example data
madElev <- fastData("madElev") # elevation raster
madForest2000 <- fastData("madForest2000") # forest raster
madForest2014 <- fastData("madForest2014") # forest raster

# Convert SpatRasters to GRasters
elev <- fast(madElev)
forest2000 <- fast(madForest2000)
forest2014 <- fast(madForest2014)

# Re-assigning values of a GRaster
constant <- elev
constant[] <- pi
names(constant) <- "pi_raster"
constant

# Re-assigning specific values of a raster
replace <- elev
replace[replace == 1] <- -20
replace

# Subsetting specific values of a raster based on another raster
elevInForest <- elev[forest2000 == 1]
plot(c(elev, forest2000, elevInForest), nr = 1)

# Adding and replacing layers of a GRaster
rasts <- c(elev, constant, forest2000)

# Combine with another layer:
add(rasts) <- forest2014 # one way
rasts

rasts <- c(rasts, forest2014) # another way

### Subsetting GRaster layers

# Subset:
rasts <- c(elev, forest2000, forest2014)
rasts[[2:3]]
subset(rasts, 2:3)
subset(rasts, c("madForest2000", "madElev"))
rasts[[c("madForest2000", "madElev")]]
rasts$madForest2000
```

```
# Get every other layer:
rasts[[c(FALSE, TRUE)]]

### Replacing layers of a GRaster

# Replace a layer
logElev <- log(elev)
names(logElev) <- "logElev"
rasts$madForest2014 <- logElev
rasts

# Replace a layer:
rasts[[3]] <- forest2000
rasts

### GVectors

# example data
madDypsis <- fastData("madDypsis") # vector of points

# Convert SpatVector to GVector
dypsis <- fast(madDypsis)

### Retrieving GVector columns

dypsis$species # Returns the column

dypsis[[c("year", "species")]] # Returns a GRaster with these columns
dypsis[ , c("year", "species")] # Same as above

### Subsetting GVector geometries

# Subset first three geometries
dypsis[1:3]
dypsis[1:3, "species"]

# Get geometries by data table condition
dypsis[dypsis$species == "Dypsis betsimisarakae"]

### (Re)assigning GVector column values

# New column
dypsis$pi <- pi

# Re-assign values
dypsis$pi <- "pie"

# Re-assign specific values
dypsis$institutionCode[dypsis$institutionCode == "MO"] <-
   "Missouri Botanical Garden"

}
```

[<- *Replace values of a GRaster*

Description

The [<- operator can be used to replace all of the values of a GRaster, or specific values depending on the expression in i. For example, you could use rast[] <- 10 to assign 10 to all cells, or rast[rast > 0] <- 10 to assign all cells with values >0 to 10. You can also use one raster to set values in another, as in rast1[rast2 > 0] <- 10.

Usage

```
## S4 replacement method for signature 'GRaster,missing,ANY'
x[i, j] <- value

## S4 replacement method for signature 'GRaster,GRaster,ANY'
x[i, j] <- value
```

Arguments

x	A GRaster.
i	Either missing or a conditional statement that resolves to a GRaster.
j	Not used
value	A numeric, integer, or logical value, or NA. Only a single value can be used.

Value

A GRaster.

Examples

```
if (grassStarted()) {

# Setup
library(terra)

### GRasters

# Example data
madElev <- fastData("madElev") # elevation raster
madForest2000 <- fastData("madForest2000") # forest raster
madForest2014 <- fastData("madForest2014") # forest raster

# Convert SpatRasters to GRasters
elev <- fast(madElev)
forest2000 <- fast(madForest2000)
forest2014 <- fast(madForest2014)
```

```
# Re-assigning values of a GRaster
constant <- elev
constant[] <- pi
names(constant) <- "pi_raster"
constant

# Re-assigning specific values of a raster
replace <- elev
replace[replace == 1] <- -20
replace

# Subsetting specific values of a raster based on another raster
elevInForest <- elev[forest2000 == 1]
plot(c(elev, forest2000, elevInForest), nr = 1)

# Adding and replacing layers of a GRaster
rasts <- c(elev, constant, forest2000)

# Combine with another layer:
add(rasts) <- forest2014 # one way
rasts

rasts <- c(rasts, forest2014) # another way

### Subsetting GRaster layers

# Subset:
rasts <- c(elev, forest2000, forest2014)
rasts[[2:3]]
subset(rasts, 2:3)
subset(rasts, c("madForest2000", "madElev"))
rasts[[c("madForest2000", "madElev")]]
rasts$madForest2000

# Get every other layer:
rasts[[c(FALSE, TRUE)]]

### Replacing layers of a GRaster

# Replace a layer
logElev <- log(elev)
names(logElev) <- "logElev"
rasts$madForest2014 <- logElev
rasts

# Replace a layer:
rasts[[3]] <- forest2000
rasts

### GVectors

# example data
madDypsis <- fastData("madDypsis") # vector of points
```

```
# Convert SpatVector to GVector
dypsis <- fast(madDypsis)

### Retrieving GVector columns

dypsis$species # Returns the column

dypsis[[c("year", "species")]] # Returns a GRaster with these columns
dypsis[ , c("year", "species")] # Same as above

### Subsetting GVector geometries

# Subset first three geometries
dypsis[1:3]
dypsis[1:3, "species"]

# Get geometries by data table condition
dypsis[dypsis$species == "Dypsis betsimisarakae"]

### (Re)assigning GVector column values

# New column
dypsis$pi <- pi

# Re-assign values
dypsis$pi <- "pie"

# Re-assign specific values
dypsis$institutionCode[dypsis$institutionCode == "MO"] <-
   "Missouri Botanical Garden"

}
```

`[[` *Subset layers from a GRaster, or specific columns from a GVector*

Description

The `[[` operator can be used to subset or remove one or more layers from a `GRaster`. It can also be used to subset or remove columns from a `GVector` with a data table.

Usage

```
## S4 method for signature 'GRaster,ANY,ANY'
x[[i, j]]

## S4 method for signature 'GVector,ANY,ANY'
x[[i, j]]
```

Arguments

x	A GRaster or GVector.
i	Numeric integer, integer, logical, or character: Indicates the layer(s) of a GRaster to subset, or the column(s) of a GVector to return. If negative numeric or integer values are supplied, then these layers or columns will be removed from the output.
j	Ignored for [[.

Value

A GRaster or GVector.

See Also

subset()

Examples

```
if (grassStarted()) {

# Setup
library(terra)

### GRasters

# Example data
madElev <- fastData("madElev") # elevation raster
madForest2000 <- fastData("madForest2000") # forest raster
madForest2014 <- fastData("madForest2014") # forest raster

# Convert SpatRasters to GRasters
elev <- fast(madElev)
forest2000 <- fast(madForest2000)
forest2014 <- fast(madForest2014)

# Re-assigning values of a GRaster
constant <- elev
constant[] <- pi
names(constant) <- "pi_raster"
constant

# Re-assigning specific values of a raster
replace <- elev
replace[replace == 1] <- -20
replace

# Subsetting specific values of a raster based on another raster
elevInForest <- elev[forest2000 == 1]
plot(c(elev, forest2000, elevInForest), nr = 1)

# Adding and replacing layers of a GRaster
```

```
rasts <- c(elev, constant, forest2000)

# Combine with another layer:
add(rasts) <- forest2014 # one way
rasts

rasts <- c(rasts, forest2014) # another way

### Subsetting GRaster layers

# Subset:
rasts <- c(elev, forest2000, forest2014)
rasts[[2:3]]
subset(rasts, 2:3)
subset(rasts, c("madForest2000", "madElev"))
rasts[[c("madForest2000", "madElev")]]
rasts$madForest2000

# Get every other layer:
rasts[[c(FALSE, TRUE)]]

### Replacing layers of a GRaster

# Replace a layer
logElev <- log(elev)
names(logElev) <- "logElev"
rasts$madForest2014 <- logElev
rasts

# Replace a layer:
rasts[[3]] <- forest2000
rasts

### GVectors

# example data
madDypsis <- fastData("madDypsis") # vector of points

# Convert SpatVector to GVector
dypsis <- fast(madDypsis)

### Retrieving GVector columns

dypsis$species # Returns the column

dypsis[[c("year", "species")]] # Returns a GRaster with these columns
dypsis[ , c("year", "species")] # Same as above

### Subsetting GVector geometries

# Subset first three geometries
dypsis[1:3]
dypsis[1:3, "species"]
```

```
# Get geometries by data table condition
dypsis[dypsis$species == "Dypsis betsimisarakae"]

### (Re)assigning GVector column values

# New column
dypsis$pi <- pi

# Re-assign values
dypsis$pi <- "pie"

# Re-assign specific values
dypsis$institutionCode[dypsis$institutionCode == "MO"] <-
   "Missouri Botanical Garden"

}
```

`[[<-` *Replace layers of a GRaster*

Description

The `[[<-` operator can be used to replace a layer in a multi-layer `GRaster`.

Usage

```
## S4 replacement method for signature 'GRaster,ANY'
x[[i]] <- value
```

Arguments

`x`	A `GRaster`.
`i`	A numeric integer, integer, logical, or character: Indicates the layer to replace. If a logical vector, then the vector must have the same length as there are layers in `x`.
`value`	Either a `GRaster` or `NULL`: If `NULL`, then the layer indicated by `i` will be removed.

Examples

```
if (grassStarted()) {

# Setup
library(terra)

### GRasters

# Example data
madElev <- fastData("madElev") # elevation raster
```

```
madForest2000 <- fastData("madForest2000") # forest raster
madForest2014 <- fastData("madForest2014") # forest raster

# Convert SpatRasters to GRasters
elev <- fast(madElev)
forest2000 <- fast(madForest2000)
forest2014 <- fast(madForest2014)

# Re-assigning values of a GRaster
constant <- elev
constant[] <- pi
names(constant) <- "pi_raster"
constant

# Re-assigning specific values of a raster
replace <- elev
replace[replace == 1] <- -20
replace

# Subsetting specific values of a raster based on another raster
elevInForest <- elev[forest2000 == 1]
plot(c(elev, forest2000, elevInForest), nr = 1)

# Adding and replacing layers of a GRaster
rasts <- c(elev, constant, forest2000)

# Combine with another layer:
add(rasts) <- forest2014 # one way
rasts

rasts <- c(rasts, forest2014) # another way

### Subsetting GRaster layers

# Subset:
rasts <- c(elev, forest2000, forest2014)
rasts[[2:3]]
subset(rasts, 2:3)
subset(rasts, c("madForest2000", "madElev"))
rasts[[c("madForest2000", "madElev")]]
rasts$madForest2000

# Get every other layer:
rasts[[c(FALSE, TRUE)]]

### Replacing layers of a GRaster

# Replace a layer
logElev <- log(elev)
names(logElev) <- "logElev"
rasts$madForest2014 <- logElev
rasts
```

```
# Replace a layer:
rasts[[3]] <- forest2000
rasts

### GVectors

# example data
madDypsis <- fastData("madDypsis") # vector of points

# Convert SpatVector to GVector
dypsis <- fast(madDypsis)

### Retrieving GVector columns

dypsis$species # Returns the column

dypsis[[c("year", "species")]] # Returns a GRaster with these columns
dypsis[ , c("year", "species")] # Same as above

### Subsetting GVector geometries

# Subset first three geometries
dypsis[1:3]
dypsis[1:3, "species"]

# Get geometries by data table condition
dypsis[dypsis$species == "Dypsis betsimisarakae"]

### (Re)assigning GVector column values

# New column
dypsis$pi <- pi

# Re-assign values
dypsis$pi <- "pie"

# Re-assign specific values
dypsis$institutionCode[dypsis$institutionCode == "MO"] <-
   "Missouri Botanical Garden"

}
```

`$` *Subset a GRaster layer, or return values from a column of a GVector's table*

Description

The `dollar` notation can be used to get a single layer of a multi-layer `GRaster` or the values of a column from a `GVector`'s data table.

Usage

```
## S4 method for signature 'GRaster'
x$name

## S4 method for signature 'GVector'
x$name
```

Arguments

x	A GRaster or GVector.
name	Character: The name of a GRaster or of a column of a GVector's data table. Names of rasters and vectors can be found using names().

Value

A GRaster or vector of the same type as the GVector's column.

See Also

subset(), [, [[

Examples

```
if (grassStarted()) {

# Setup
library(terra)

### GRasters

# Example data
madElev <- fastData("madElev") # elevation raster
madForest2000 <- fastData("madForest2000") # forest raster
madForest2014 <- fastData("madForest2014") # forest raster

# Convert SpatRasters to GRasters
elev <- fast(madElev)
forest2000 <- fast(madForest2000)
forest2014 <- fast(madForest2014)

# Re-assigning values of a GRaster
constant <- elev
constant[] <- pi
names(constant) <- "pi_raster"
constant

# Re-assigning specific values of a raster
replace <- elev
replace[replace == 1] <- -20
replace
```

```
# Subsetting specific values of a raster based on another raster
elevInForest <- elev[forest2000 == 1]
plot(c(elev, forest2000, elevInForest), nr = 1)

# Adding and replacing layers of a GRaster
rasts <- c(elev, constant, forest2000)

# Combine with another layer:
add(rasts) <- forest2014 # one way
rasts

rasts <- c(rasts, forest2014) # another way

### Subsetting GRaster layers

# Subset:
rasts <- c(elev, forest2000, forest2014)
rasts[[2:3]]
subset(rasts, 2:3)
subset(rasts, c("madForest2000", "madElev"))
rasts[[c("madForest2000", "madElev")]]
rasts$madForest2000

# Get every other layer:
rasts[[c(FALSE, TRUE)]]

### Replacing layers of a GRaster

# Replace a layer
logElev <- log(elev)
names(logElev) <- "logElev"
rasts$madForest2014 <- logElev
rasts

# Replace a layer:
rasts[[3]] <- forest2000
rasts

### GVectors

# example data
madDypsis <- fastData("madDypsis") # vector of points

# Convert SpatVector to GVector
dypsis <- fast(madDypsis)

### Retrieving GVector columns

dypsis$species # Returns the column

dypsis[[c("year", "species")]] # Returns a GRaster with these columns
dypsis[ , c("year", "species")] # Same as above
```

```
### Subsetting GVector geometries

# Subset first three geometries
dypsis[1:3]
dypsis[1:3, "species"]

# Get geometries by data table condition
dypsis[dypsis$species == "Dypsis betsimisarakae"]

### (Re)assigning GVector column values

# New column
dypsis$pi <- pi

# Re-assign values
dypsis$pi <- "pie"

# Re-assign specific values
dypsis$institutionCode[dypsis$institutionCode == "MO"] <-
   "Missouri Botanical Garden"

}
```

`$<-` *Replace a raster layer or a column from a vector's data table*

Description

The `$<-` notation can be used to replace a specific layer in a multi-layer `GRaster`, or a to replace a specific column from a `GVector`'s data table.

Usage

```
## S4 replacement method for signature 'GRaster'
x$name <- value

## S4 replacement method for signature 'GVector'
x$name <- value
```

Arguments

`x`	A `GRaster` or `GVector`.
`name`	Character: Name of the `GRaster` layer to replace, or name of the `GVector` column to replace.
`value`	Character: The name of a `GRaster` layer or the name of a column in a `GVector`'s data table. Names of rasters and vector tables' columns cab be obtained using `names()`.

Value

A GRaster or the column of a GVector.

See Also

$, [[<-, and add<-

Examples

```
if (grassStarted()) {

# Setup
library(terra)

### GRasters

# Example data
madElev <- fastData("madElev") # elevation raster
madForest2000 <- fastData("madForest2000") # forest raster
madForest2014 <- fastData("madForest2014") # forest raster

# Convert SpatRasters to GRasters
elev <- fast(madElev)
forest2000 <- fast(madForest2000)
forest2014 <- fast(madForest2014)

# Re-assigning values of a GRaster
constant <- elev
constant[] <- pi
names(constant) <- "pi_raster"
constant

# Re-assigning specific values of a raster
replace <- elev
replace[replace == 1] <- -20
replace

# Subsetting specific values of a raster based on another raster
elevInForest <- elev[forest2000 == 1]
plot(c(elev, forest2000, elevInForest), nr = 1)

# Adding and replacing layers of a GRaster
rasts <- c(elev, constant, forest2000)

# Combine with another layer:
add(rasts) <- forest2014 # one way
rasts

rasts <- c(rasts, forest2014) # another way

### Subsetting GRaster layers
```

```
# Subset:
rasts <- c(elev, forest2000, forest2014)
rasts[[2:3]]
subset(rasts, 2:3)
subset(rasts, c("madForest2000", "madElev"))
rasts[[c("madForest2000", "madElev")]]
rasts$madForest2000

# Get every other layer:
rasts[[c(FALSE, TRUE)]]

### Replacing layers of a GRaster

# Replace a layer
logElev <- log(elev)
names(logElev) <- "logElev"
rasts$madForest2014 <- logElev
rasts

# Replace a layer:
rasts[[3]] <- forest2000
rasts

### GVectors

# example data
madDypsis <- fastData("madDypsis") # vector of points

# Convert SpatVector to GVector
dypsis <- fast(madDypsis)

### Retrieving GVector columns

dypsis$species # Returns the column

dypsis[[c("year", "species")]] # Returns a GRaster with these columns
dypsis[ , c("year", "species")] # Same as above

### Subsetting GVector geometries

# Subset first three geometries
dypsis[1:3]
dypsis[1:3, "species"]

# Get geometries by data table condition
dypsis[dypsis$species == "Dypsis betsimisarakae"]

### (Re)assigning GVector column values

# New column
dypsis$pi <- pi

# Re-assign values
```

```
    dypsis$pi <- "pie"

    # Re-assign specific values
    dypsis$institutionCode[dypsis$institutionCode == "MO"] <-
       "Missouri Botanical Garden"

    }
```

Index