Package 'gdalraster'

November 19, 2025

Title Bindings to 'GDAL'

Version 2.3.0

Description API bindings to the Geospatial Data Abstraction Library ('GDAL', <https://gdal.org>). Implements the 'GDAL' Raster and Vector Data Models. Bindings are implemented with 'Rcpp' modules. Exposed C++ classes and stand-alone functions wrap much of the 'GDAL' API and provide additional functionality. Calling signatures resemble the native C, C++ and Python APIs provided by the 'GDAL' project. Class 'GDALRaster' encapsulates a 'GDALDataset' and its raster band objects. Class 'GDALVector' encapsulates an 'OGRLayer' and the 'GDALDataset' that contains it. Initial bindings are provided to the unified 'gdal' command line interface added in 'GDAL' 3.11. C++ stand-alone functions provide bindings to most 'GDAL' ``traditional'' raster and vector utilities, including 'OGR' facilities for vector geoprocessing, several algorithms, as well as the Geometry API ('GEOS' via 'GDAL' headers), the Spatial Reference Systems API, and methods for coordinate transformation. Bindings to the Virtual Systems Interface ('VSI') API implement standard file system operations abstracted for URLs, cloud storage services, 'Zip'/'GZip'/'7z'/'RAR', in-memory files, as well as regular local file systems. This provides a single interface for operating on file system objects that works the same for any storage backend. A custom raster calculator evaluates a user-defined R expression on a layer or stack of layers, with pixel x/y available as variables in the expression. Raster 'combine()' identifies and counts unique pixel combinations across multiple input layers, with optional raster output of the pixel-level combination IDs. Basic plotting capability is provided for raster and vector display. 'gdalraster' leans toward minimalism and the use of simple, lightweight objects for holding raw data. Currently, only minimal S3 class interfaces have been implemented for selected R objects that contain spatial data. 'gdalraster' may be useful in applications that need scalable, low-level I/O, or prefer a direct 'GDAL' API.

License MIT + file LICENSE

Copyright See file inst/COPYRIGHTS for details.

URL https://usdaforestservice.github.io/gdalraster/, https://github.com/USDAForestService/gdalraster

BugReports https://github.com/USDAForestService/gdalraster/issues

Depends R (>= 4.2.0)

Imports bit64, graphics, grDevices, methods, nanoarrow, Rcpp (>= 1.0.7), stats, tools, utils, wk, xml2, yyjsonr

LinkingTo nanoarrow, Rcpp, RcppInt64

Suggests gt, knitr, rmarkdown, scales, testthat (>= 3.0.0)

NeedsCompilation yes

SystemRequirements C++17, GDAL (>= 3.1.0, built against GEOS)

Encoding UTF-8

RoxygenNote 7.3.3

VignetteBuilder knitr

Config/testthat/edition 3

Author Chris Toney [aut, cre] (R interface/additional functionality),
Michael D. Sumner [ctb],
Frank Warmerdam [ctb, cph] (GDAL API documentation; src/progress_r.cpp from gdal/port/cpl_progress.cpp),
Even Rouault [ctb, cph] (GDAL API documentation),
Marius Appel [ctb, cph] (configure.ac based on https://github.com/appelmar/gdalcubes),
Daniel James [ctb, cph] (Boost combine hashes method in src/cmb_table.h),
Peter Dimov [ctb, cph] (Boost combine hashes method in src/cmb_table.h)

Maintainer Chris Toney <jctoney@gmail.com>

Repository CRAN

Date/Publication 2025-11-19 08:20:07 UTC

Contents

`gdalraster-package` *Bindings to the GDAL API*

Description

gdalraster is an interface to the Geospatial Data Abstraction Library (GDAL) providing an R implementation of the GDAL Raster and Vector Data Models. Bindings also include the GDAL Geometry API, Spatial Reference Systems API, utilities and algorithms, methods for coordinate transformation, and the Virtual Systems Interface (VSI) API. Calling signatures resemble those of the native C, C++ and Python APIs provided by the GDAL project. See https://gdal.org/en/stable/api/ for details of the GDAL API.

Details

Experimental bindings to the modernized `gdal` Command Line Interface (CLI) are implemented in `GDALAlg-class` and related convenience functions: `gdal_commands()`, `gdal_usage()`, `gdal_run()`, `gdal_alg()` (*Requires GDAL >= 3.11.3*)

Core raster functionality is contained in class `GDALRaster` and several related stand-alone functions:

- `GDALRaster-class` is an exposed C++ class that allows opening a raster dataset and calling methods on the `GDALDataset`, `GDALDriver` and `GDALRasterBand` objects in the underlying API (e.g., get/set parameters, read/write pixel data).
- raster creation: `create()`, `createCopy()`, `rasterFromRaster()`, `translate()`, `getCreationOptions()`, `validateCreationOptions()`
- virtual raster: `autoCreateWarpedVRT()`, `buildVRT()`, `rasterToVRT()`
- algorithms/utilities: `dem_proc()`, `fillNodata()`, `footprint()`, `make_chunk_index()`, `polygonize()`, `rasterize()`, `sieveFilter()`, `warp()`, `GDALRaster$getChecksum()`
- raster attribute tables: `buildRAT()`, `displayRAT()`, `GDALRaster$getDefaultRAT()`, `GDALRaster$setDefaultRAT()`
- multidimensional raster: `mdim_as_classic()`, `mdim_info()`, `mdim_translate()`
- geotransform conversion: `apply_geotransform()`, `get_pixel_line()`, `inv_geotransform()`, `pixel_extract()`

- data type convenience functions: dt_size(), dt_is_complex(), dt_is_integer(), dt_is_floating(), dt_is_signed(), dt_union(), dt_union_with_value(), dt_find(), dt_find_for_value()

Core vector functionality is contained in class GDALVector and several related stand-alone functions:

- GDALVector-class is an exposed C++ class that allows opening a vector dataset and calling methods on a specified OGRLayer object that it contains (e.g., obtain layer information, set attribute and/or spatial filters, read/write feature data).
- OGR vector utilities: ogrinfo(), ogr2ogr(), ogr_reproject(), ogr_define, ogr_manage, ogr_proc()

Bindings to the GDAL Geometry API, Spatial Reference Systems API, methods for coordinate transformation, the Virtual Systems Interface (VSI) API, general data management and system configuration are implemented in several stand-alone functions:

- Geometry API: bbox_from_wkt(), bbox_to_wkt(), bbox_intersect(), bbox_union(), bbox_transform(), g_factory, g_wk2wk(), g_query, g_util, g_binary_pred, g_binary_op, g_unary_op, g_measures, g_coords(), g_envelope(), g_transform(), geos_version(), plot_geom()
- Spatial Reference Systems API: srs_convert, srs_query
- coordinate transformation: transform_xy(), inv_project(), transform_bounds()
- data management: addFilesInZip(), copyDatasetFiles(), deleteDataset(), renameDataset(), bandCopyWholeRaster(), identifyDriver(), inspectDataset()
- Virtual Systems Interface API: VSIFile-class, vsi_clear_path_options(), vsi_copy_file(), vsi_curl_clear_cache(), vsi_get_disk_free_space(), vsi_get_file_metadata(), vsi_get_fs_options(), vsi_get_fs_prefixes(), vsi_is_local(), vsi_mkdir(), vsi_read_dir(), vsi_rename(), vsi_rmdir(), vsi_set_path_option(), vsi_stat(), vsi_supports_rnd_write(), vsi_supports_seq_write(), vsi_sync(), vsi_unlink(), vsi_unlink_batch()
- GDAL configuration: gdal_version, gdal_compute_version(), gdal_formats(), gdal_get_driver_md(), get_cache_used(), get_cache_max(), set_cache_max(), get_config_option(), set_config_option(), get_num_cpus(), get_usable_physical_ram(), has_spatialite(), http_enabled(), push_error_handler(), pop_error_handler(), dump_open_datasets()
- PROJ configuration: proj_version(), proj_search_paths(), proj_networking()

Additional functionality includes:

- RunningStats-class calculates mean and variance in one pass. The min, max, sum, and count are also tracked (efficient summary statistics on data streams).
- CmbTable-class implements a hash table for counting unique combinations of integer values.
- combine() overlays multiple rasters so that a unique ID is assigned to each unique combination of input values. Pixel counts for each unique combination are obtained, and combination IDs are optionally written to an output raster.
- calc() evaluates an R expression for each pixel in a raster layer or stack of layers. Individual pixel coordinates are available as variables in the R expression, as either x/y in the raster projected coordinate system or inverse projected longitude/latitude.
- plot_raster() displays raster data using base R graphics. Supports single-band grayscale, RGB, color tables and color map functions (e.g., color ramp).

Note

Documentation for the API bindings borrows heavily from the GDAL documentation, (c) 1998-2025, Frank Warmerdam, Even Rouault, and others, MIT license.

Sample datasets included with the package are used in examples throughout the documentation. The sample data sources include:

- LANDFIRE raster layers describing terrain, vegetation and wildland fuels (LF 2020 version)
- Landsat C2 Analysis Ready Data from USGS Earth Explorer
- Monitoring Trends in Burn Severity (MTBS) fire perimeters from 1984-2022
- NLCD Tree Canopy Cover produced by USDA Forest Service
- National Park Service Open Data vector layers for roads and points-of-interest
- Montana State Library boundary layer for Yellowstone National Park

Metadata for these sample datasets are in inst/extdata/metadata.zip and inst/extdata/ynp_features.zip.

`system.file()` is used in the examples to access the sample datasets. This enables the code to run regardless of where R is installed. Users will normally give file names as a regular full path or relative to the current working directory.

Temporary files are created in some examples which have cleanup code wrapped in `dontshow`. While the cleanup code is not shown in the documentation, note that this code runs by default if examples are run with `example()`.

Author(s)

GDAL is by: Frank Warmerdam, Even Rouault and others
(see https://github.com/OSGeo/gdal/graphs/contributors)

R interface/additional functionality: Chris Toney

Maintainer: Chris Toney <jctoney at gmail.com>

See Also

GDAL Raster Data Model:
https://gdal.org/en/stable/user/raster_data_model.html

Raster driver descriptions:
https://gdal.org/en/stable/drivers/raster/index.html

Geotransform tutorial:
https://gdal.org/en/stable/tutorials/geotransforms_tut.html

GDAL Vector Data Model:
https://gdal.org/en/stable/user/vector_data_model.html

Vector driver descriptions:
https://gdal.org/en/stable/drivers/vector/index.html

GDAL Virtual File Systems:
https://gdal.org/en/stable/user/virtual_file_systems.html

addFilesInZip *Create/append to a potentially Seek-Optimized ZIP file (SOZip)*

Description

`addFilesInZip()` will create new or open existing ZIP file, and add one or more compressed files potentially using the seek optimization extension. This function is basically a wrapper for `CPLAddFileInZip()` in the GDAL Common Portability Library, but optionally creates a new ZIP file first (with `CPLCreateZip()`). It provides a subset of functionality in the GDAL `sozip` command-line utility (https://gdal.org/en/stable/programs/sozip.html). Requires GDAL >= 3.7.

Usage

```
addFilesInZip(
  zip_file,
  add_files,
  overwrite = FALSE,
  full_paths = TRUE,
  sozip_enabled = NULL,
  sozip_chunk_size = NULL,
  sozip_min_file_size = NULL,
  num_threads = NULL,
  content_type = NULL,
  quiet = FALSE
)
```

Arguments

`zip_file`	Filename of the ZIP file. Will be created if it does not exist or if `overwrite = TRUE`. Otherwise will append to an existing file.
`add_files`	Character vector of one or more input filenames to add.
`overwrite`	Logical scalar. Overwrite the target zip file if it already exists.
`full_paths`	Logical scalar. By default, the full path will be stored (relative to the current directory). `FALSE` to store just the name of a saved file (drop the path).
`sozip_enabled`	String. Whether to generate a SOZip index for the file. One of `"AUTO"` (the default), `"YES"` or `"NO"` (see Details).
`sozip_chunk_size`	The chunk size for a seek-optimized file. Defaults to 32768 bytes. The value is specified in bytes, or K and M suffix can be used respectively to specify a value in kilo-bytes or mega-bytes. Will be coerced to string.
`sozip_min_file_size`	The minimum file size to decide if a file should be seek-optimized, in `sozip_enabled="AUTO"` mode. Defaults to 1 MB byte. The value is specified in bytes, or K, M or G suffix can be used respectively to specify a value in kilo-bytes, mega-bytes or giga-bytes. Will be coerced to string.

num_threads	Number of threads used for SOZip generation. Defaults to "ALL_CPUS" or specify an integer value (coerced to string).
content_type	String Content-Type value for the file. This is stored as a key-value pair in the extra field extension 'KV' (0x564b) dedicated to storing key-value pair metadata.
quiet	Logical scalar. TRUE for quiet mode, no progress messages emitted. Defaults to FALSE.

Details

A Seek-Optimized ZIP file (SOZip) contains one or more compressed files organized and annotated such that a SOZip-aware reader can perform very fast random access within the .zip file (see https://github.com/sozip/sozip-spec). Large compressed files can be accessed directly from SOZip without prior decompression. The .zip file is otherwise fully backward compatible.

If sozip_enabled="AUTO" (the default), a file is seek-optimized only if its size is above the values of sozip_min_file_size (default 1 MB) and sozip_chunk_size (default 32768). In "YES" mode, all input files will be seek-optimized. In "NO" mode, no input files will be seek-optimized. The default can be changed with the CPL_SOZIP_ENABLED configuration option.

Value

Logical indicating success (invisible TRUE). An error is raised if the operation fails.

Note

The GDAL_NUM_THREADS configuration option can be set to ALL_CPUS or an integer value to specify the number of threads to use for SOZip-compressed files (see set_config_option()).

SOZip can be validated with:

```
vsi_get_file_metadata(zip_file, domain="ZIP")
```

where zip_file uses the /vsizip/ prefix.

See Also

vsi_get_file_metadata()

Examples

```
lcp_file <- system.file("extdata/storm_lake.lcp", package="gdalraster")
zip_file <- file.path(tempdir(), "storml_lcp.zip")

# Requires GDAL >= 3.7
if (gdal_version_num() >= gdal_compute_version(3, 7, 0)) {
  addFilesInZip(zip_file, lcp_file, full_paths = FALSE,
                sozip_enabled = "YES", num_threads = 1)

  print("Files in zip archive:")
  print(unzip(zip_file, list = TRUE))
```

```
  # Open with GDAL using Virtual File System handler '/vsizip/'
  # see: https://gdal.org/en/stable/user/virtual_file_systems.html#vsizip-zip-archives
  lcp_in_zip <- file.path("/vsizip", zip_file, "storm_lake.lcp")
  print("SOZip metadata:")
  print(vsi_get_file_metadata(lcp_in_zip, domain = "ZIP"))

  ds <- new(GDALRaster, lcp_in_zip)
  ds$info()
  ds$close()

}
```

---

apply_geotransform *Apply geotransform (raster column/row to geospatial x/y)*

Description

apply_geotransform() applies geotransform coefficients to raster coordinates in pixel/line space (column/row), converting into georeferenced (x/y) coordinates. Wrapper of GDALApplyGeoTransform() in the GDAL API, operating on matrix input.

Usage

```
apply_geotransform(col_row, gt)
```

Arguments

col_row	Numeric matrix of raster column, row (pixel/line) coordinates (or two-column data frame that will be coerced to numeric matrix, or a vector of column, row for one coordinate).
gt	Either a numeric vector of length six containing the affine geotransform for the raster, or an object of class GDALRaster from which the geotransform will be obtained.

Value

Numeric matrix of geospatial x/y coordinates.

Note

Bounds checking on the input coordinates is done if gt is obtained from an object of class GDALRaster. See Note for get_pixel_line().

See Also

GDALRaster$getGeoTransform(), get_pixel_line()

Examples

```
raster_file <- system.file("extdata/storm_lake.lcp", package="gdalraster")
ds <- new(GDALRaster, raster_file)

# compute some raster coordinates in column/row space
set.seed(42)
col_coords <- runif(10, min = 0, max = ds$getRasterXSize() - 0.00001)
row_coords <- runif(10, min = 0, max = ds$getRasterYSize() - 0.00001)
col_row <- cbind(col_coords, row_coords)

# convert to geospatial x/y coordinates
gt <- ds$getGeoTransform()
apply_geotransform(col_row, gt)

# or, using the class method
ds$apply_geotransform(col_row)

# bounds checking
col_row <- rbind(col_row, c(ds$getRasterXSize(), ds$getRasterYSize()))
ds$apply_geotransform(col_row)

ds$close()
```

---

autoCreateWarpedVRT *Create a virtual warped dataset automatically*

Description

`autoCreateWarpedVRT()` creates a warped virtual dataset representing the input raster warped into a target coordinate system. The output virtual dataset will be "north-up" in the target coordinate system. GDAL automatically determines the bounds and resolution of the output virtual raster which should be large enough to include all the input raster. Wrapper of `GDALAutoCreateWarpedVRT()` in the GDAL Warper API.

Usage

```
autoCreateWarpedVRT(
  src_ds,
  dst_wkt,
  resample_alg,
  src_wkt = "",
  max_err = 0,
  alpha_band = FALSE
)
```

Arguments

`src_ds`	An object of class `GDALRaster` for the source dataset.

dst_wkt	WKT string specifying the coordinate system to convert to. If empty string ("") no change of coordinate system will take place.
resample_alg	Character string specifying the sampling method to use. One of NearestNeighbour, Bilinear, Cubic, CubicSpline, Lanczos, Average, RMS or Mode.
src_wkt	WKT string specifying the coordinate system of the source raster. If empty string it will be read from the source raster (the default).
max_err	Numeric scalar specifying the maximum error measured in input pixels that is allowed in approximating the transformation (0.0 for exact calculations, the default).
alpha_band	Logical scalar, TRUE to create an alpha band if the source dataset has none. Defaults to FALSE.

Value

An object of class GDALRaster for the new virtual dataset. An error is raised if the operation fails.

Note

The returned dataset will have no associated filename for itself. If you want to write the virtual dataset to a VRT file, use the $setFilename() method on the returned GDALRaster object to assign a filename before it is closed.

Examples

```
elev_file <- system.file("extdata/storml_elev.tif", package="gdalraster")
ds <- new(GDALRaster, elev_file)

ds2 <- autoCreateWarpedVRT(ds, epsg_to_wkt(5070), "Bilinear")
ds2$info()

## set filename before close if a VRT file is needed for the virtual dataset
# ds2$setFilename("/path/to/file.vrt")

ds2$close()
ds$close()
```

---

bandCopyWholeRaster *Copy a whole raster band efficiently*

Description

bandCopyWholeRaster() copies the complete raster contents of one band to another similarly configured band. The source and destination bands must have the same xsize and ysize. The bands do not have to have the same data type. It implements efficient copying, in particular "chunking" the copy in substantial blocks. This is a wrapper for GDALRasterBandCopyWholeRaster() in the GDAL API.

Usage

```
bandCopyWholeRaster(
  src_filename,
  src_band,
  dst_filename,
  dst_band,
  options = NULL,
  quiet = FALSE
)
```

Arguments

src_filename	Filename of the source raster.
src_band	Band number in the source raster to be copied.
dst_filename	Filename of the destination raster.
dst_band	Band number in the destination raster to copy into.
options	Optional list of transfer hints in a vector of "NAME=VALUE" pairs. The currently supported options are: • "COMPRESSED=YES" to force alignment on target dataset block sizes to achieve best compression. • "SKIP_HOLES=YES" to skip chunks that contain only empty blocks. Empty blocks are blocks that are generally not physically present in the file, and when read through GDAL, contain only pixels whose value is the nodata value when it is set, or whose value is 0 when the nodata value is not set. The query is done in an efficient way without reading the actual pixel values (if implemented by the raster format driver, otherwise will not be skipped).
quiet	Logical scalar. If TRUE, a progress bar will not be displayed. Defaults to FALSE.

Value

Logical indicating success (invisible TRUE). An error is raised if the operation fails.

See Also

GDALRaster-class, create(), createCopy(), rasterFromRaster()

Examples

```
## copy Landsat data from a single-band file to a new multi-band image
b5_file <- system.file("extdata/sr_b5_20200829.tif", package="gdalraster")
dst_file <- file.path(tempdir(), "sr_multi.tif")
rasterFromRaster(b5_file, dst_file, nbands = 7, init = 0)
opt <- c("COMPRESSED=YES", "SKIP_HOLES=YES")
bandCopyWholeRaster(b5_file, 1, dst_file, 5, options = opt)
ds <- new(GDALRaster, dst_file)
ds$getStatistics(band = 5, approx_ok = FALSE, force = TRUE)
ds$close()
```

bbox_from_wkt *Get the bounding box of a geometry specified in OGC WKT format*

Description

bbox_from_wkt() returns the bounding box of a WKT 2D geometry (e.g., LINE, POLYGON, MULTIPOLYGON).

Usage

```
bbox_from_wkt(wkt, extend_x = 0, extend_y = 0)
```

Arguments

wkt	Character. OGC WKT string for a simple feature 2D geometry.
extend_x	Numeric scalar. Distance to extend the output bounding box in both directions along the x-axis (results in xmin = bbox[1] - extend_x, xmax = bbox[3] + extend_x).
extend_y	Numeric scalar. Distance to extend the output bounding box in both directions along the y-axis (results in ymin = bbox[2] - extend_y, ymax = bbox[4] + extend_y).

Value

Numeric vector of length four containing the xmin, ymin, xmax, ymax of the geometry specified by wkt (possibly extended by values in extend_x, extend_y).

See Also

bbox_to_wkt()

Examples

```
bnd <- "POLYGON ((324467.3 5104814.2, 323909.4 5104365.4, 323794.2
5103455.8, 324970.7 5102885.8, 326420.0 5103595.3, 326389.6 5104747.5,
325298.1 5104929.4, 325298.1 5104929.4, 324467.3 5104814.2))"
bbox_from_wkt(bnd, 100, 100)
```

bbox_intersect *Bounding box intersection / union*

Description

bbox_intersect() returns the bounding box intersection, and bbox_union() returns the bounding box union, for input of either raster file names or list of bounding boxes. All of the inputs must be in the same projected coordinate system.

Usage

```
bbox_intersect(x, as_wkt = FALSE)

bbox_union(x, as_wkt = FALSE)
```

Arguments

x	Either a character vector of raster file names, or a list with each element a bounding box numeric vector (xmin, ymin, xmax, ymax).
as_wkt	Logical. TRUE to return the bounding box as a polygon in OGC WKT format, or FALSE to return as a numeric vector.

Value

The intersection (bbox_intersect()) or union (bbox_union()) of inputs. If as_wkt = FALSE, a numeric vector of length four containing xmin, ymin, xmax, ymax. If as_wkt = TRUE, a character string containing OGC WKT for the bbox as POLYGON.

See Also

bbox_from_wkt(), bbox_to_wkt()

Examples

```
bbox_list <-list()

elev_file <- system.file("extdata/storml_elev.tif", package="gdalraster")
ds <- new(GDALRaster, elev_file)
bbox_list[[1]] <- ds$bbox()
ds$close()

b5_file <- system.file("extdata/sr_b5_20200829.tif", package="gdalraster")
ds <- new(GDALRaster, b5_file)
bbox_list[[2]] <- ds$bbox()
ds$close()

bnd <- "POLYGON ((324467.3 5104814.2, 323909.4 5104365.4, 323794.2
5103455.8, 324970.7 5102885.8, 326420.0 5103595.3, 326389.6 5104747.5,
```

```
325298.1 5104929.4, 325298.1 5104929.4, 324467.3 5104814.2))"
bbox_list[[3]] <- bbox_from_wkt(bnd)

print(bbox_list)
bbox_intersect(bbox_list)
bbox_union(bbox_list)
```

---

bbox_to_wkt *Convert a bounding box to POLYGON in OGC WKT format*

Description

`bbox_to_wkt()` returns a WKT POLYGON string for the given bounding box.

Usage

```
bbox_to_wkt(bbox, extend_x = 0, extend_y = 0)
```

Arguments

`bbox`	Numeric vector of length four containing xmin, ymin, xmax, ymax.
`extend_x`	Numeric scalar. Distance in units of `bbox` to extend the rectangle in both directions along the x-axis (results in `xmin = bbox[1] - extend_x`, `xmax = bbox[3] + extend_x`).
`extend_y`	Numeric scalar. Distance in units of `bbox` to extend the rectangle in both directions along the y-axis (results in `ymin = bbox[2] - extend_y`, `ymax = bbox[4] + extend_y`).

Value

Character string for an OGC WKT polygon.

See Also

`bbox_from_wkt()`, `g_buffer()`

Examples

```
elev_file <- system.file("extdata/storml_elev.tif", package="gdalraster")
ds <- new(GDALRaster, elev_file, read_only=TRUE)
bbox_to_wkt(ds$bbox())
ds$close()
```

bbox_transform *Transform a bounding box to a different projection*

Description

bbox_transform() is a convenience function to transform the coordinates of a boundary from their current spatial reference system to a new target spatial reference system.

Usage

```
bbox_transform(bbox, srs_from, srs_to, use_transform_bounds = TRUE)
```

Arguments

bbox	Numeric vector of length four containing a bounding box (xmin, ymin, xmax, ymax) to transform.
srs_from	Character string specifying the spatial reference system for pts. May be in WKT format or any of the formats supported by srs_to_wkt().
srs_to	Character string specifying the output spatial reference system. May be in WKT format or any of the formats supported by srs_to_wkt().
use_transform_bounds	Logical value, TRUE to use transform_bounds() (the default, requires GDAL >= 3.4). If FALSE, transformation is done with g_transform().

Details

With use_transform_bounds = TRUE (the default) this function returns:

```
# requires GDAL >= 3.4
transform_bounds(bbox, srs_from, srs_to)
```

See Details for transform_bounds() for cases where the bounds crossed the antimeridian.

With use_transform_bounds = FALSE, this function returns:

```
bbox_to_wkt(bbox) |>
  g_transform(srs_from, srs_to) |>
  bbox_from_wkt()
```

See the Note for g_transform() for cases where the bounds crossed the antimeridian.

Value

Numeric vector of length four containing a transformed bounding box (xmin, ymin, xmax, ymax).

See Also

bbox_from_wkt(), g_transform(), transform_bounds()

Examples

```
bb <- c(-1405880.72, -1371213.76, 5405880.72, 5371213.76)

# the default assumes GDAL >= 3.4
if (gdal_version_num() >= gdal_compute_version(3, 4, 0)) {
  bb_wgs84 <- bbox_transform(bb, "EPSG:32661", "EPSG:4326")
} else {
  bb_wgs84 <- bbox_transform(bb, "EPSG:32661", "EPSG:4326",
                             use_transform_bounds = FALSE)
}

print(bb_wgs84)
```

---

buildRAT *Build a GDAL Raster Attribute Table with VALUE, COUNT*

Description

`buildRAT()` reads all pixels of an input raster to obtain the set of unique values and their counts. The result is returned as a data frame suitable for use with the class method `GDALRaster$setDefaultRAT()`. The returned data frame might be further modified before setting as a Raster Attribute Table in a dataset, for example, by adding columns containing class names, color values, or other information (see Details). An optional input data frame containing such attributes may be given, in which case `buildRAT()` will attempt to join the additional columns and automatically assign the appropriate metadata on the output data frame (i.e., assign R attributes on the data frame and its columns that define usage in a GDAL Raster Attribute Table).

Usage

```
buildRAT(
  raster,
  band = 1L,
  col_names = c("VALUE", "COUNT"),
  table_type = "athematic",
  na_value = NULL,
  join_df = NULL,
  quiet = FALSE
)
```

Arguments

`raster`	Either a `GDALRaster` object, or a character string containing the file name of a raster dataset to open.
`band`	Integer scalar, band number to read (default `1L`).
`col_names`	Character vector of length two containing names to use for column 1 (pixel values) and column 2 (pixel counts) in the output data frame (defaults are `c("VALUE", "COUNT")`).

table_type	Character string describing the type of the attribute table. One of either "thematic", or "athematic" for continuous data (the default).
na_value	Numeric scalar. If the set of unique pixel values has an NA, it will be recoded to na_value in the returned data frame. If NULL (the default), NA will not be recoded.
join_df	Optional data frame for joining additional attributes. Must have a column of unique values with the same name as col_names[1] ("VALUE" by default).
quiet	Logical scalar. If TRUE``, a progress bar will not be displayed. Defaults to FALSE".

Details

A GDAL Raster Attribute Table (or RAT) provides attribute information about pixel values. Raster attribute tables can be used to represent histograms, color tables, and classification information. Each row in the table applies to either a single pixel value or a range of values, and might have attributes such as the histogram count for that value (or range), the color that pixels of that value (or range) should be displayed, names of classes, or various other information.

Each column in a raster attribute table has a name, a type (integer, double, or string), and a GDALRATFieldUsage. The usage distinguishes columns with particular understood purposes (such as color, histogram count, class name), and columns that have other purposes not understood by the library (long labels, ancillary attributes, etc).

In the general case, each row has a field indicating the minimum pixel value falling into that category, and a field indicating the maximum pixel value. In the GDAL API, these are indicated with usage values of GFU_Min and GFU_Max. In the common case where each row is a discrete pixel value, a single column with usage GFU_MinMax would be used instead. In R, the table is represented as a data frame with column attribute "GFU" containing the field usage as a string, e.g., "Max", "Min" or "MinMax" (case-sensitive). The full set of possible field usage descriptors is:

GFU attr	GDAL enum	Description
"Generic"	GFU_Generic	General purpose field
"PixelCount"	GFU_PixelCount	Histogram pixel count
"Name"	GFU_Name	Class name
"Min"	GFU_Min	Class range minimum
"Max"	GFU_Max	Class range maximum
"MinMax"	GFU_MinMax	Class value (min=max)
"Red"	GFU_Red	Red class color (0-255)
"Green"	GFU_Green	Green class color (0-255)
"Blue"	GFU_Blue	Blue class color (0-255)
"Alpha"	GFU_Alpha	Alpha transparency (0-255)
"RedMin"	GFU_RedMin	Color range red minimum
"GreenMin"	GFU_GreenMin	Color range green minimum
"BlueMin"	GFU_BlueMin	Color range blue minimum
"AlphaMin"	GFU_AlphaMin	Color range alpha minimum
"RedMax"	GFU_RedMax	Color range red maximum
"GreenMax"	GFU_GreenMax	Color range green maximum
"BlueMax"	GFU_BlueMax	Color range blue maximum
"AlphaMax"	GFU_AlphaMax	Color range alpha maximum

buildRAT() assigns GFU "MinMax" on the column of pixel values (named "VALUE" by default) and GFU "PixelCount" on the column of counts (named "COUNT" by default). If join_df is given, the additional columns that result from joining will have GFU assigned automatically based on the column names (*ignoring case*). First, the additional column names are checked for containing the string "name" (e.g., "classname", "TypeName", "EVT_NAME", etc). The first matching column (if any) will be assigned a GFU of "Name" (=GFU_Name, the field usage descriptor for class names). Next, columns named "R" or "Red" will be assigned GFU "Red", columns named "G" or "Green" will be assigned GFU "Green", columns named "B" or "Blue" will be assigned GFU "Blue", and columns named "A" or "Alpha" will be assigned GFU "Alpha". Finally, any remaining columns that have not been assigned a GFU will be assigned "Generic".

In a variation of RAT, all the categories are of equal size and regularly spaced, and the categorization can be determined by knowing the value at which the categories start and the size of a category. This is called "Linear Binning" and the information is kept specially on the raster attribute table as a whole. In R, a RAT that uses linear binning would have the following attributes set on the data frame: attribute "Row0Min" = the numeric lower bound (pixel value) of the first category, and attribute "BinSize" = the numeric width of each category (in pixel value units). buildRAT() does not create tables with linear binning, but one could be created manually based on the specifications above, and applied to a raster with the class method GDALRaster$setDefaultRAT().

A raster attribute table is thematic or athematic (continuous). In R, this is defined by an attribute on the data frame named "GDALRATTableType" with value of either "thematic" or "athematic".

Value

A data frame with at least two columns containing the set of unique pixel values and their counts. These columns have attribute "GFU" set to "MinMax" for the values, and "PixelCount" for the counts. If join_df is given, the returned data frame will have additional columns that result from merge(). The "GFU" attribute of the additional columns will be assigned automatically based on the column names (*case-insensitive* matching, see Details). The returned data frame has attribute "GDALRATTableType" set to table_type.

Note

The full raster will be scanned.

If na_value is not specified, then an NA pixel value (if present) will not be recoded in the output data frame. This may have implications if joining to other data (NA will not match), or when using the returned data frame to set a default RAT on a dataset (NA will be interpreted as the value that R uses internally to represent it for the type, e.g., -2147483648 for NA_integer_). In some cases, removing the row in the output data frame with value NA, rather than recoding, may be desirable (i.e., by removing manually or by side effect of joining via merge(), for example). Users should consider what is appropriate for a particular case.

See Also

GDALRaster$getDefaultRAT(), GDALRaster$setDefaultRAT(), displayRAT()

vignette("raster-attribute-tables")

Examples

```
evt_file <- system.file("extdata/storml_evt.tif", package="gdalraster")
# make a copy to modify
f <- file.path(tempdir(), "storml_evt_tmp.tif")
file.copy(evt_file,  f)

ds <- new(GDALRaster, f, read_only=FALSE)
ds$getDefaultRAT(band = 1) # NULL

# get the full attribute table for LANDFIRE EVT from the CSV file
evt_csv <- system.file("extdata/LF20_EVT_220.csv", package="gdalraster")
evt_df <- read.csv(evt_csv)
nrow(evt_df)
head(evt_df)
evt_df <- evt_df[,1:7]

tbl <- buildRAT(ds,
                table_type = "thematic",
                na_value = -9999,
                join_df = evt_df)

nrow(tbl)
head(tbl)

# attributes on the data frame and its columns define usage in a GDAL RAT
attributes(tbl)
attributes(tbl$VALUE)
attributes(tbl$COUNT)
attributes(tbl$EVT_NAME)
attributes(tbl$EVT_LF)
attributes(tbl$EVT_PHYS)
attributes(tbl$R)
attributes(tbl$G)
attributes(tbl$B)

ds$setDefaultRAT(band = 1, tbl)
ds$flushCache()

tbl2 <- ds$getDefaultRAT(band = 1)
nrow(tbl2)
head(tbl2)

ds$close()


# Display
evt_gt <- displayRAT(tbl2, title = "Storm Lake EVT Raster Attribute Table")
class(evt_gt)  # an object of class "gt_tbl" from package gt
# To show the table:
# evt_gt
# or simply call `displayRAT()` as above but without assignment
# `vignette("raster-attribute-tables")` has example output
```

buildVRT *Build a GDAL virtual raster from a list of datasets*

Description

buildVRT() is a wrapper of the gdalbuildvrt command-line utility for building a VRT (Virtual Dataset) that is a mosaic of the list of input GDAL datasets (see https://gdal.org/en/stable/programs/gdalbuildvrt.html).

Usage

```
buildVRT(vrt_filename, input_rasters, cl_arg = NULL, quiet = FALSE)
```

Arguments

vrt_filename	Character string. Filename of the output VRT.
input_rasters	Character vector of input raster filenames.
cl_arg	Optional character vector of command-line arguments to gdalbuildvrt.
quiet	Logical scalar. If TRUE, a progress bar will not be displayed. Defaults to FALSE.

Details

Several command-line options are described in the GDAL documentation at the URL above. By default, the input files are considered as tiles of a larger mosaic and the VRT file has as many bands as one of the input files. Alternatively, the -separate argument can be used to put each input raster into a separate band in the VRT dataset.

Some amount of checks are done to assure that all files that will be put in the resulting VRT have similar characteristics: number of bands, projection, color interpretation.... If not, files that do not match the common characteristics will be skipped. (This is true in the default mode for virtual mosaicing, and not when using the -separate option).

In a virtual mosaic, if there is spatial overlap between input rasters then the order of files appearing in the list of sources matter: files that are listed at the end are the ones from which the data will be fetched. Note that nodata will be taken into account to potentially fetch data from less priority datasets.

Value

Logical indicating success (invisible TRUE). An error is raised if the operation fails.

See Also

rasterToVRT()

Examples

```
# build a virtual 3-band RGB raster from individual Landsat band files
b4_file <- system.file("extdata/sr_b4_20200829.tif", package="gdalraster")
b5_file <- system.file("extdata/sr_b5_20200829.tif", package="gdalraster")
b6_file <- system.file("extdata/sr_b6_20200829.tif", package="gdalraster")
band_files <- c(b6_file, b5_file, b4_file)
vrt_file <- file.path(tempdir(), "storml_b6_b5_b4.vrt")
buildVRT(vrt_file, band_files, cl_arg = "-separate")
ds <- new(GDALRaster, vrt_file)
ds$getRasterCount()
plot_raster(ds, nbands = 3, main = "Landsat 6-5-4 (vegetative analysis)")
ds$close()
```

`calc` *Raster calculation*

Description

`calc()` evaluates an R expression for each pixel in a raster layer or stack of layers. Each layer is defined by a raster filename, band number, and a variable name to use in the R expression. If not specified, band defaults to 1 for each input raster. Variable names default to `LETTERS` if not specified (`A` (layer 1), `B` (layer 2), ...). All of the input layers must have the same extent and cell size. The projection will be read from the first raster in the list of inputs. Individual pixel coordinates are also available as variables in the R expression, as either x/y in the raster projected coordinate system or inverse projected longitude/latitude. Multiband output is supported as of gdalraster 1.11.0.

Usage

```
calc(
  expr,
  rasterfiles,
  bands = NULL,
  var.names = NULL,
  dstfile = tempfile("rastcalc", fileext = ".tif"),
  fmt = NULL,
  dtName = "Int16",
  out_band = NULL,
  options = NULL,
  nodata_value = NULL,
  setRasterNodataValue = FALSE,
  usePixelLonLat = NULL,
  write_mode = "safe",
  quiet = FALSE
)
```

Arguments

expr	An R expression as a character string (e.g., "A + B").
rasterfiles	Character vector of source raster filenames.
bands	Integer vector of band numbers to use for each raster layer.
var.names	Character vector of variable names to use for each raster layer.
dstfile	Character filename of output raster.
fmt	Output raster format name (e.g., "GTiff" or "HFA"). Will attempt to guess from the output filename if not specified.
dtName	Character name of output data type (e.g., Byte, Int16, UInt16, Int32, UInt32, Float32).
out_band	Integer band number(s) in dstfile for writing output. Defaults to 1. Multiband output is supported as of gdalraster 1.11.0, in which case out_band would be a vector of band numbers.
options	Optional list of format-specific creation options in a vector of "NAME=VALUE" pairs (e.g., options = c("COMPRESS=LZW") to set LZW compression during creation of a GTiff file).
nodata_value	Numeric value to assign if expr returns NA.
setRasterNodataValue	Logical. TRUE will attempt to set the raster format nodata value to nodata_value, or FALSE not to set a raster nodata value.
usePixelLonLat	This argument is deprecated and will be removed in a future version. Variable names pixelLon and pixelLat can be used in expr, and the pixel x/y coordinates will be inverse projected to longitude/latitude (adds computation time).
write_mode	Character. Name of the file write mode for output. One of: • safe - execution stops if dstfile already exists (no output written) • overwrite - if dstfile exists if will be overwritten with a new file • update - if dstfile exists, will attempt to open in update mode and write output to out_band
quiet	Logical scalar. If TRUE, a progress bar will not be displayed. Defaults to FALSE.

Details

The variables in expr are vectors of length raster xsize (row vectors of the input raster layer(s)). The expression should return a vector also of length raster xsize (an output row). Four special variable names are available in expr: pixelX and pixelY provide pixel center coordinates in projection units. pixelLon and pixelLat can also be used, in which case the pixel x/y coordinates will be inverse projected to longitude/latitude (in the same geographic coordinate system used by the input projection, which is read from the first input raster). Note that inverse projection adds computation time.

To refer to specific bands in a multi-band input file, repeat the filename in rasterfiles and specify corresponding band numbers in bands, along with optional variable names in var.names, for example,

```
rasterfiles = c("multiband.tif", "multiband.tif")
bands = c(4, 5)
var.names = c("B4", "B5")
```

Output will be written to `dstfile`. To update a file that already exists, set `write_mode = "update"` and set `out_band` to an existing band number(s) in `dstfile` (new bands cannot be created in `dstfile`).

To write multiband output, `expr` must return a vector of values interleaved by band. This is equivalent to, and can also be returned as, a matrix `m` with `nrow(m)` equal to `length()` of an input vector, and `ncol(m)` equal to the number of output bands. In matrix form, each column contains a vector of output values for a band. `length(m)` must be equal to the `length()` of an input vector multiplied by `length(out_band)`. The dimensions described above are assumed and not read from the return value of `expr`.

Value

Returns the output filename invisibly.

See Also

GDALRaster-class, combine(), rasterToVRT()

Examples

```
## Using pixel longitude/latitude

# Hopkins bioclimatic index (HI) as described in:
# Bechtold, 2004, West. J. Appl. For. 19(4):245-251.
# Integrates elevation, latitude and longitude into an index of the
# phenological occurrence of springtime. Here it is relativized to
# mean values for an eight-state region in the western US.
# Positive HI means spring is delayed by that number of days relative
# to the reference position, while negative values indicate spring is
# advanced. The original equation had elevation units as feet, so
# converting m to ft in `expr`.

elev_file <- system.file("extdata/storml_elev.tif", package="gdalraster")

# expression to calculate HI
expr <- "round( ((ELEV_M * 3.281 - 5449) / 100) +
                ((pixelLat - 42.16) * 4) +
                ((-116.39 - pixelLon) * 1.25) )"

# calc() writes to a tempfile by default
hi_file <- calc(expr = expr,
                rasterfiles = elev_file,
                var.names = "ELEV_M",
                dtName = "Int16",
                nodata_value = -32767,
                setRasterNodataValue = TRUE)
```

```
ds <- new(GDALRaster, hi_file)
# min, max, mean, sd
ds$getStatistics(band = 1, approx_ok = FALSE, force = TRUE)
ds$close()



## Calculate normalized difference vegetation index (NDVI)

# Landast band 4 (red) and band 5 (near infrared):
b4_file <- system.file("extdata/sr_b4_20200829.tif", package="gdalraster")
b5_file <- system.file("extdata/sr_b5_20200829.tif", package="gdalraster")

expr <- "((B5 * 0.0000275 - 0.2) - (B4 * 0.0000275 - 0.2)) /
         ((B5 * 0.0000275 - 0.2) + (B4 * 0.0000275 - 0.2))"
ndvi_file <- calc(expr = expr,
                  rasterfiles = c(b4_file, b5_file),
                  var.names = c("B4", "B5"),
                  dtName = "Float32",
                  nodata_value = -32767,
                  setRasterNodataValue = TRUE)

ds <- new(GDALRaster, ndvi_file)
ds$getStatistics(band=1, approx_ok=FALSE, force=TRUE)
ds$close()



## Reclassify a variable by rule set

# Combine two raster layers and look for specific combinations. Then
# recode to a new value by rule set.
#
# Based on example in:
#   Stratton, R.D. 2009. Guidebook on LANDFIRE fuels data acquisition,
#   critique, modification, maintenance, and model calibration.
#   Gen. Tech. Rep. RMRS-GTR-220. U.S. Department of Agriculture,
#   Forest Service, Rocky Mountain Research Station. 54 p.
# Context: Refine national-scale fuels data to improve fire simulation
#   results in localized applications.
# Issue: Areas with steep slopes (40+ degrees) were mapped as
#   GR1 (101; short, sparse dry climate grass) and
#   GR2 (102; low load, dry climate grass) but were not carrying fire.
# Resolution: After viewing these areas in Google Earth,
#   NB9 (99; bare ground) was selected as the replacement fuel model.

# look for combinations of slope >= 40 and FBFM 101 or 102
lcp_file <- system.file("extdata/storm_lake.lcp", package="gdalraster")
rasterfiles <- c(lcp_file, lcp_file)
var.names <- c("SLP", "FBFM")
bands <- c(2, 4)
tbl <- combine(rasterfiles, var.names, bands)
nrow(tbl)
```

```
tbl_subset <- subset(tbl, SLP >= 40 & FBFM %in% c(101,102))
print(tbl_subset)       # twelve combinations meet the criteria
sum(tbl_subset$count)   # 85 total pixels

# recode these pixels to 99 (bare ground)
# the LCP driver does not support in-place write so make a copy as GTiff
tif_file <- file.path(tempdir(), "storml_lndscp.tif")
createCopy("GTiff", tif_file, lcp_file)

expr <- "ifelse( SLP >= 40 & FBFM %in% c(101,102), 99, FBFM)"
calc(expr = expr,
     rasterfiles = c(lcp_file, lcp_file),
     bands = c(2, 4),
     var.names = c("SLP", "FBFM"),
     dstfile = tif_file,
     out_band = 4,
     write_mode = "update")

# verify the ouput
rasterfiles <- c(tif_file, tif_file)
tbl <- combine(rasterfiles, var.names, bands)
tbl_subset <- subset(tbl, SLP >= 40 & FBFM %in% c(101,102))
print(tbl_subset)
sum(tbl_subset$count)

# if LCP file format is needed:
# createCopy("LCP", "storml_edited.lcp", tif_file)
```

CmbTable-class *Class for counting unique combinations of integers*

Description

CmbTable implements a hash table having a vector of integers as the key, and the count of occurrences of each unique integer combination as the value. A unique ID is assigned to each unique combination of input values.

CmbTable is a C++ class exposed directly to R (via RCPP_EXPOSED_CLASS). Methods of the class are accessed using the $ operator. **Note that all arguments to class methods are required and must be given in the order documented.** Naming the arguments is optional but may be preferred for readability.

Arguments

keyLen	The number of integer values comprising each combination.
varNames	Optional character vector of names for the variables in the combination.

Value

An object of class `CmbTable`. Contains a hash table having a vector of `keyLen` integers as the key, and the count of occurrences of each unique integer combination as the value. Class methods that operate on the hash table are described in Details.

Usage (see Details)

```
## Constructors
cmb <- new(CmbTable, keyLen)
# or, giving the variable names:
cmb <- new(CmbTable, keyLen, varNames)

## Methods
cmb$update(int_cmb, incr)
cmb$updateFromMatrix(int_cmbs, incr)
cmb$updateFromMatrixByRow(int_cmbs, incr)
cmb$asDataFrame()
cmb$asMatrix()
```

Details

Constructors:

`new(CmbTable, keyLen)`
Default variable names will be assigned as `V1`, `V2`, ... Returns an object of class `CmbTable`.

`new(CmbTable, keyLen, varNames)`
Alternate constructor to specify variable names. Returns an object of class `CmbTable`.

Methods:

`$update(int_cmb, incr)`
Updates the hash table for the integer combination in the numeric vector `int_cmb` (will be coerced to integer by truncation). If this combination exists in the table, its count will be incremented by `incr`. If the combination is not found in the table, it will be inserted with count set to `incr`. The caller should ensure that the length of the input vector is equal to the key length (`keyLen`) when using this method. Returns the unique ID assigned to this combination. Combination IDs are sequential whole numbers starting at `1`.

`$updateFromMatrix(int_cmbs, incr)`
This method is the same as `$update()` but for a numeric matrix of integer combinations `int_cmbs` (coerced to integer by truncation). The matrix is arranged with each column vector forming an integer combination. For example, the rows of the matrix could be one row each from a set of `keyLen` rasters all read at the same extent and pixel resolution (i.e., row-by-row raster overlay). The method calls `$update()` on each combination (each column of `int_cmbs`), incrementing count by `incr` for existing combinations, or inserting new combinations with count set to `incr`. Returns a numeric vector of length `ncol(int_cmbs)` containing the IDs assigned to the combinations.

`$updateFromMatrixByRow(int_cmbs, incr)`
This method is the same as `$updateFromMatrix()` above except the integer combinations are in rows of the matrix `int_cmbs` (columns are the variables). The method calls `$update()` on

each combination (each row of int_cmbs), incrementing count by incr for existing combinations, or inserting new combinations with count set to incr. Returns a numeric vector of length nrow(int_cmbs) containing the IDs assigned to the combinations.

$asDataFrame()
Returns the CmbTable as a data frame with column cmbid containing the unique combination IDs, column count containing the counts of occurrences, and keyLen columns (with names from varNames) containing the integer values comprising each unique combination.

$asMatrix()
Returns the CmbTable as a matrix with column 1 (cmbid) containing the unique combination IDs, column 2 (count) containing the counts of occurrences, and columns 3:keyLen+2 (with names from varNames) containing the integer values comprising each unique combination.

Examples

```
m <- matrix(c(1,2,3,1,2,3,4,5,6,1,3,2,4,5,6,1,1,1), 3, 6, byrow = FALSE)
rownames(m) <- c("layer1", "layer2", "layer3")
print(m)

cmb <- new(CmbTable, 3, rownames(m))
cmb

cmb$updateFromMatrix(m, 1)
cmb$asDataFrame()

cmb$update(c(4,5,6), 1)
cmb$update(c(1,3,5), 1)
cmb$asDataFrame()

# same as above but matrix arranged with integer combinations in the rows
m <- matrix(c(1,2,3,1,2,3,4,5,6,1,3,2,4,5,6,1,1,1), 6, 3, byrow = TRUE)
colnames(m) <- c("V1", "V2", "V3")
print(m)

cmb <- new(CmbTable, 3)
cmb$updateFromMatrixByRow(m, 1)
cmb$asDataFrame()

cmb$update(c(4,5,6), 1)
cmb$update(c(1,3,5), 1)
cmb$asDataFrame()
```

---

combine *Raster overlay for unique combinations*

Description

combine() overlays multiple rasters so that a unique ID is assigned to each unique combination of input values. The input raster layers typically have integer data types (floating point will be coerced to integer by truncation), and must have the same projection, extent and cell size. Pixel counts

for each unique combination are obtained, and combination IDs are optionally written to an output raster.

Usage

```
combine(
  rasterfiles,
  var.names = NULL,
  bands = NULL,
  dstfile = NULL,
  fmt = NULL,
  dtName = "UInt32",
  options = NULL,
  quiet = FALSE
)
```

Arguments

rasterfiles	Character vector of raster filenames to combine.
var.names	Character vector of length(rasterfiles) containing variable names for each raster layer. Defaults will be assigned if var.names are omitted.
bands	Numeric vector of length(rasterfiles) containing the band number to use for each raster in rasterfiles. Band 1 will be used for each input raster if bands are not specified.
dstfile	Character. Optional output raster filename for writing the per-pixel combination IDs. The output raster will be created (and overwritten if it already exists).
fmt	Character. Output raster format name (e.g., "GTiff" or "HFA").
dtName	Character. Output raster data type name. Combination IDs are sequential integers starting at 1. The data type for the output raster should be large enough to accommodate the potential number of unique combinations of the input values (e.g., "UInt16" or the default "UInt32").
options	Optional list of format-specific creation options in a vector of "NAME=VALUE" pairs (e.g., options = c("COMPRESS=LZW") to set LZW compression during creation of a GTiff file).
quiet	Logical scalar. If TRUE, progress bar and messages will be suppressed. Defaults to FALSE.

Details

To specify input raster layers that are bands of a multi-band raster file, repeat the filename in rasterfiles and provide the corresponding band numbers in bands. For example:

```
rasterfiles <- c("multi-band.tif", "multi-band.tif", "other.tif")
bands <- c(4, 5, 1)
var.names <- c("multi_b4", "multi_b5", "other")
```

rasterToVRT() provides options for virtual clipping, resampling and pixel alignment, which may be helpful here if the input rasters are not already aligned on a common extent and cell size.

If an output raster of combination IDs is written, the user should verify that the number of combinations obtained did not exceed the range of the output data type. Combination IDs are sequential integers starting at 1. Typical output data types are the unsigned types: Byte (0 to 255), UInt16 (0 to 65535) and UInt32 (the default, 0 to 4294967295).

Value

A data frame with column cmbid containing the combination IDs, column count containing the pixel counts for each combination, and length(rasterfiles) columns named var.names containing the integer values comprising each unique combination.

See Also

CmbTable-class, GDALRaster-class, calc(), rasterToVRT()

buildRAT() to compute a table of the unique pixel values and their counts for a single raster layer

Examples

```
evt_file <- system.file("extdata/storml_evt.tif", package="gdalraster")
evc_file <- system.file("extdata/storml_evc.tif", package="gdalraster")
evh_file <- system.file("extdata/storml_evh.tif", package="gdalraster")
rasterfiles <- c(evt_file, evc_file, evh_file)
var.names <- c("veg_type", "veg_cov", "veg_ht")
tbl <- combine(rasterfiles, var.names)
nrow(tbl)
tbl <- tbl[order(-tbl$count),]
head(tbl, n = 20)

# combine two bands from a multi-band file and write the combination IDs
# to an output raster
lcp_file <- system.file("extdata/storm_lake.lcp", package="gdalraster")
rasterfiles <- c(lcp_file, lcp_file)
bands <- c(4, 5)
var.names <- c("fbfm", "tree_cov")
cmb_file <- file.path(tempdir(), "fbfm_cov_cmbid.tif")
opt <- c("COMPRESS=LZW")
tbl <- combine(rasterfiles, var.names, bands, cmb_file, options = opt)
head(tbl)
ds <- new(GDALRaster, cmb_file)
ds$info()
ds$close()
```

copyDatasetFiles *Copy the files of a dataset*

Description

copyDatasetFiles() copies all the files associated with a dataset. Wrapper for GDALCopyDatasetFiles() in the GDAL API.

Usage

```
copyDatasetFiles(new_filename, old_filename, format = "")
```

Arguments

new_filename	New name for the dataset (copied to).
old_filename	Old name for the dataset (copied from).
format	Raster format short name (e.g., "GTiff"). If set to empty string "" (the default), will attempt to guess the raster format from old_filename.

Value

Logical TRUE if no error or FALSE on failure.

Note

If format is set to an empty string "" (the default) then the function will try to identify the driver from old_filename. This is done internally in GDAL by invoking the Identify method of each registered GDALDriver in turn. The first driver that successful identifies the file name will be returned. An error is raised if a format cannot be determined from the passed file name.

See Also

GDALRaster-class, create(), createCopy(), deleteDataset(), renameDataset(), vsi_copy_file()

Examples

```
lcp_file <- system.file("extdata/storm_lake.lcp", package="gdalraster")
ds <- new(GDALRaster, lcp_file)
ds$getFileList()
ds$close()

lcp_tmp <- file.path(tempdir(), "storm_lake_copy.lcp")
copyDatasetFiles(lcp_tmp, lcp_file)
ds_copy <- new(GDALRaster, lcp_tmp)
ds_copy$getFileList()
ds_copy$close()

deleteDataset(lcp_tmp)
```

create *Create a new uninitialized raster*

Description

`create()` makes an empty raster in the specified format.

Usage

```
create(
  format,
  dst_filename,
  xsize,
  ysize,
  nbands,
  dataType,
  options = NULL,
  return_obj = FALSE
)
```

Arguments

`format`	Character string giving a raster format short name (e.g., `"GTiff"`).
`dst_filename`	Character string giving the filename to create.
`xsize`	Integer width of raster in pixels.
`ysize`	Integer height of raster in pixels.
`nbands`	Integer number of bands.
`dataType`	Character string containing the data type name. (e.g., common data types include Byte, Int16, UInt16, Int32, Float32).
`options`	Optional list of format-specific creation options in a character vector of `"NAME=VALUE"` pairs (e.g., `options = c("COMPRESS=LZW")` to set LZW compression during creation of a GTiff file). The APPEND_SUBDATASET=YES option can be specified to avoid prior destruction of existing dataset.
`return_obj`	Logical scalar. If `TRUE`, an object of class `GDALRaster` opened on the newly created dataset will be returned, otherwise returns a logical value. Defaults to `FALSE`.

Value

By default, returns a logical value indicating success (invisible `TRUE`, output written to `dst_filename`). An error is raised if the operation fails. An object of class `GDALRaster` open on the output dataset will be returned if `return_obj = TRUE`.

Note

dst_filename may be an empty string ("") with format = "MEM" and return_obj = TRUE to create an In-memory Raster (https://gdal.org/en/stable/drivers/raster/mem.html).

See Also

GDALRaster-class, createCopy(), getCreationOptions(), rasterFromRaster()

Examples

```
new_file <- file.path(tempdir(), "newdata.tif")
ds <- create(format = "GTiff",
             dst_filename = new_file,
             xsize = 143,
             ysize = 107,
             nbands = 1,
             dataType = "Int16",
             return_obj = TRUE)

# EPSG:26912 - NAD83 / UTM zone 12N
ds$setProjection(epsg_to_wkt(26912))

gt <- c(323476, 30, 0, 5105082, 0, -30)
ds$setGeoTransform(gt)

ds$setNoDataValue(band = 1, -9999)
ds$fillRaster(band = 1, -9999, 0)

# ...

# close the dataset when done
ds$close()
```

---

createColorRamp *Create a color ramp*

Description

createColorRamp() is a wrapper for GDALCreateColorRamp() in the GDAL API. It automatically creates a color ramp from one color entry to another. Output is an integer matrix in color table format for use with GDALRaster$setColorTable().

Usage

```
createColorRamp(
  start_index,
  start_color,
  end_index,
```

```
  end_color,
  palette_interp = "RGB"
)
```

Arguments

start_index	Integer start index (raster value).
start_color	Integer vector of length three or four. A color entry value to start the ramp (e.g., RGB values).
end_index	Integer end index (raster value).
end_color	Integer vector of length three or four. A color entry value to end the ramp (e.g., RGB values).
palette_interp	One of "Gray", "RGB" (the default), "CMYK" or "HLS" describing interpretation of start_color and end_color values (see GDAL Color Table).

Value

Integer matrix with five columns containing the color ramp from start_index to end_index, with raster index values in column 1 and color entries in columns 2:5).

Note

createColorRamp() could be called several times, using rbind() to combine multiple ramps into the same color table. Possible duplicate rows in the resulting table are not a problem when used in GDALRaster$setColorTable() (i.e., when end_color of one ramp is the same as start_color of the next ramp).

See Also

GDALRaster$getColorTable(), GDALRaster$getPaletteInterp()

Examples

```
# create a color ramp for tree canopy cover percent
# band 5 of an LCP file contains canopy cover
lcp_file <- system.file("extdata/storm_lake.lcp", package="gdalraster")
ds <- new(GDALRaster, lcp_file)
ds$getDescription(band=5)
ds$getMetadata(band=5, domain="")
ds$close()

# create a GTiff file with Byte data type for the canopy cover band
# recode nodata -9999 to 255
tcc_file <- calc(expr = "ifelse(CANCOV == -9999, 255, CANCOV)",
                 rasterfiles = lcp_file,
                 bands = 5,
                 var.names = "CANCOV",
                 fmt = "GTiff",
                 dtName = "Byte",
                 nodata_value = 255,
```

```
                setRasterNodataValue = TRUE)

ds_tcc <- new(GDALRaster, tcc_file, read_only=FALSE)

# create a color ramp from 0 to 100 and set as the color table
colors <- createColorRamp(start_index = 0,
                          start_color = c(211, 211, 211),
                          end_index = 100,
                          end_color = c(0, 100, 0))

print(colors)
ds_tcc$setColorTable(band=1, col_tbl=colors, palette_interp = "RGB")
ds_tcc$setRasterColorInterp(band = 1, col_interp = "Palette")

# close and re-open the dataset in read_only mode
ds_tcc$open(read_only=TRUE)

plot_raster(ds_tcc, interpolate = FALSE, legend = TRUE,
            main = "Storm Lake Tree Canopy Cover (%)")
ds_tcc$close()
```

`createCopy` *Create a copy of a raster*

Description

`createCopy()` copies a raster dataset, optionally changing the format. The extent, cell size, number of bands, data type, projection, and geotransform are all copied from the source raster.

Usage

```
createCopy(
  format,
  dst_filename,
  src_filename,
  strict = FALSE,
  options = NULL,
  quiet = FALSE,
  return_obj = FALSE
)
```

Arguments

`format`	Character string giving the format short name for the output raster (e.g., `"GTiff"`).
`dst_filename`	Character string giving the filename to create.
`src_filename`	Either a character string giving the filename of the source raster, or object of class `GDALRaster` for the source.

strict	Logical. TRUE if the copy must be strictly equivalent, or more normally FALSE (the default) indicating that the copy may adapt as needed for the output format.
options	Optional list of format-specific creation options in a vector of "NAME=VALUE" pairs (e.g., options = c("COMPRESS=LZW") to set LZW compression during creation of a GTiff file). The APPEND_SUBDATASET=YES option can be specified to avoid prior destruction of existing dataset.
quiet	Logical scalar. If TRUE, a progress bar will be not be displayed. Defaults to FALSE.
return_obj	Logical scalar. If TRUE, an object of class GDALRaster opened on the newly created dataset will be returned. Defaults to FALSE.

Value

By default, returns a logical value indicating success (invisible TRUE, output written to dst_filename). An error is raised if the operation fails. An object of class GDALRaster open on the output dataset will be returned if return_obj = TRUE.

Note

dst_filename may be an empty string ("") with format = "MEM" and return_obj = TRUE to create an In-memory Raster (https://gdal.org/en/stable/drivers/raster/mem.html).

See Also

GDALRaster-class, create(), getCreationOptions(), rasterFromRaster(), translate()

Examples

```
lcp_file <- system.file("extdata/storm_lake.lcp", package="gdalraster")
tif_file <- file.path(tempdir(), "storml_lndscp.tif")
ds <- createCopy(format = "GTiff",
                 dst_filename = tif_file,
                 src_filename = lcp_file,
                 options = "COMPRESS=LZW",
                 return_obj = TRUE)

ds$getMetadata(band = 0, domain = "IMAGE_STRUCTURE")

for (band in 1:ds$getRasterCount())
    ds$setNoDataValue(band, -9999)
ds$getStatistics(band = 1, approx_ok = FALSE, force = TRUE)

ds$close()
```

data_type_helpers *Helper functions for GDAL raster data types*

Description

These are convenience functions that return information about a raster data type, return the smallest data type that can fully express two input data types, or find the smallest data type able to support specified requirements.

Usage

```
dt_size(dt, as_bytes = TRUE)

dt_is_complex(dt)

dt_is_integer(dt)

dt_is_floating(dt)

dt_is_signed(dt)

dt_union(dt, dt_other)

dt_union_with_value(dt, value, is_complex = FALSE)

dt_find(bits, is_signed, is_floating, is_complex = FALSE)

dt_find_for_value(value, is_complex = FALSE)
```

Arguments

dt	Character string containing a GDAL data type name (e.g., "Byte", "Int16", "UInt16", "Int32", "UInt32", "Float32", "Float64", etc.)
as_bytes	Logical value, TRUE to return data type size in bytes (the default), FALSE to return the size in bits.
dt_other	Character string containing a GDAL data type name.
value	Numeric value for which to find a data type (passing the real part if is_complex = TRUE).
is_complex	Logical value, TRUE if value is complex (default is FALSE), or if complex values are necessary in dt_find().
bits	Integer value specifying the number of bits necessary.
is_signed	Logical value, TRUE if negative values are necessary.
is_floating	Logical value, TRUE if non-integer values are necessary.

Details

`dt_size()` returns the data type size in bytes by default, optionally in bits (returns zero if `dt` is not recognized).

`dt_is_complex()` returns `TRUE` if the passed type is complex (one of CInt16, CInt32, CFloat32 or CFloat64), i.e., if it consists of a real and imaginary component.

`dt_is_integer()` returns `TRUE` if the passed type is integer (one of Byte, Int16, UInt16, Int32, UInt32, CInt16, CInt32).

`dt_is_floating()` returns `TRUE` if the passed type is floating (one of Float32, Float16, Float64, CFloat16, CFloat32, CFloat64).

`dt_is_signed()` returns `TRUE` if the passed type is signed.

`dt_union()` returns the smallest data type that can fully express both input data types (returns a data type name as character string).

`dt_union_with_value()` unions a data type with the data type found for a given value, and returns the resulting data type name as character string.

`dt_find()` finds the smallest data type able to support the given requirements (returns a data type name as character string).

`dt_find_for_value()` finds the smallest data type able to support the given `value` (returns a data type name as character string).

See Also

`GDALRaster$getDataTypeName()`

Examples

```
elev_file <- system.file("extdata/storml_elev.tif", package="gdalraster")
ds <- new(GDALRaster, elev_file)

ds$getDataTypeName(band = 1) |> dt_size()
ds$getDataTypeName(band = 1) |> dt_size(as_bytes = FALSE)
ds$getDataTypeName(band = 1) |> dt_is_complex()
ds$getDataTypeName(band = 1) |> dt_is_integer()
ds$getDataTypeName(band = 1) |> dt_is_floating()
ds$getDataTypeName(band = 1) |> dt_is_signed()

ds$close()

f <- system.file("extdata/complex.tif", package="gdalraster")
ds <- new(GDALRaster, f)

ds$getDataTypeName(band = 1) |> dt_size()
ds$getDataTypeName(band = 1) |> dt_size(as_bytes = FALSE)
ds$getDataTypeName(band = 1) |> dt_is_complex()
ds$getDataTypeName(band = 1) |> dt_is_integer()
ds$getDataTypeName(band = 1) |> dt_is_floating()
ds$getDataTypeName(band = 1) |> dt_is_signed()

ds$close()
```

```
dt_union("Byte", "Int16")
dt_union_with_value("Byte", -1)
dt_union_with_value("Byte", 256)

dt_find(bits = 32, is_signed = FALSE, is_floating = FALSE)
dt_find_for_value(0)
dt_find_for_value(-1)
dt_find_for_value(NaN)
dt_find_for_value(.Machine$integer.max)
```

DEFAULT_DEM_PROC *List of default DEM processing options*

Description

These values are used in `dem_proc()` as the default processing options:

```
list(
     "hillshade" =    c("-z", "1", "-s", "1", "-az", "315",
                        "-alt", "45", "-alg", "Horn",
                        "-combined", "-compute_edges"),
     "slope" =        c("-s", "1", "-alg", "Horn", "-compute_edges"),
     "aspect" =       c("-alg", "Horn", "-compute_edges"),
     "color-relief" = character(),
     "TRI" =          c("-alg", "Riley", "-compute_edges"),
     "TPI" =          c("-compute_edges"),
     "roughness" =    c("-compute_edges"))
```

Usage

```
DEFAULT_DEM_PROC
```

Format

An object of class `list` of length 7.

See Also

`dem_proc()`

https://gdal.org/en/stable/programs/gdaldem.html for a description of all available command-line options for each processing mode

DEFAULT_NODATA *List of default nodata values by raster data type*

Description

These values are currently used in gdalraster when a nodata value is needed but has not been specified:

```
list("Byte" = 255, "Int8" = -128,
     "UInt16" = 65535, "Int16" = -32767,
     "UInt32" = 4294967293, "Int32" = -2147483647,
     "Float32" = -99999.0, "Float64" = -99999.0)
```

Usage

```
DEFAULT_NODATA
```

Format

An object of class list of length 8.

deleteDataset *Delete named dataset*

Description

deleteDataset() will attempt to delete the named dataset in a format specific fashion. Full featured drivers will delete all associated files, database objects, or whatever is appropriate. The default behavior when no format specific behavior is provided is to attempt to delete all the files that would be returned by GDALRaster$getFileList() on the dataset. The named dataset should not be open in any existing GDALRaster objects when deleteDataset() is called. Wrapper for GDALDeleteDataset() in the GDAL API.

Usage

```
deleteDataset(filename, format = "")
```

Arguments

filename	Filename to delete (should not be open in a GDALRaster object).
format	Raster format short name (e.g., "GTiff"). If set to empty string "" (the default), will attempt to guess the raster format from filename.

Value

Logical TRUE if no error or FALSE on failure.

Note

If `format` is set to an empty string `""` (the default) then the function will try to identify the driver from `filename`. This is done internally in GDAL by invoking the `Identify` method of each registered `GDALDriver` in turn. The first driver that successful identifies the file name will be returned. An error is raised if a format cannot be determined from the passed file name.

See Also

GDALRaster-class, create(), createCopy(), copyDatasetFiles(), renameDataset()

Examples

```
b5_file <- system.file("extdata/sr_b5_20200829.tif", package="gdalraster")
b5_tmp <- file.path(tempdir(), "b5_tmp.tif")
file.copy(b5_file,  b5_tmp)

ds <- new(GDALRaster, b5_tmp)
ds$buildOverviews("BILINEAR", levels = c(2, 4, 8), bands = c(1))
files <- ds$getFileList()
print(files)
ds$close()
file.exists(files)
deleteDataset(b5_tmp)
file.exists(files)
```

---

dem_proc *GDAL DEM processing*

Description

`dem_proc()` generates DEM derivatives from an input elevation raster. This function is a wrapper for the `gdaldem` command-line utility. See https://gdal.org/en/stable/programs/gdaldem.html for details.

Usage

```
dem_proc(
  mode,
  srcfile,
  dstfile,
  mode_options = DEFAULT_DEM_PROC[[mode]],
  color_file = NULL,
  quiet = FALSE
)
```

Arguments

mode	Character. Name of the DEM processing mode. One of hillshade, slope, aspect, color-relief, TRI, TPI or roughness.
srcfile	Filename of the source elevation raster.
dstfile	Filename of the output raster.
mode_options	An optional character vector of command-line options (see DEFAULT_DEM_PROC for default values).
color_file	Filename of a text file containing lines formatted as: "elevation_value red green blue". Only used when mode = "color-relief".
quiet	Logical scalar. If TRUE, a progress bar will not be displayed. Defaults to FALSE.

Value

Logical indicating success (invisible TRUE). An error is raised if the operation fails.

Note

Band 1 of the source elevation raster is read by default, but this can be changed by including a -b command-line argument in mode_options. See the documentation for gdaldem for a description of all available options for each processing mode.

Examples

```
## hillshade
elev_file <- system.file("extdata/storml_elev.tif", package="gdalraster")
out_file <- file.path(tempdir(), "storml_hillshade.tif")
dem_proc("hillshade", elev_file, out_file)

ds <- new(GDALRaster, out_file)
plot_raster(ds)

ds$close()
```

---

displayRAT *Display a GDAL Raster Attribute Table*

Description

displayRAT() generates a presentation table. Colors are shown if the Raster Attribute Table contains RGB columns. This function requires package gt.

Usage

```
displayRAT(tbl, title = "Raster Attribute Table")
```

Arguments

tbl	A data frame formatted as a GDAL RAT (e.g., as returned by buildRAT() or GDALRaster$getDefaultRAT()).
title	Character string to be used in the table title.

Value

An object of class "gt_tbl" (i.e., a table created with gt::gt()).

See Also

buildRAT(), GDALRaster$getDefaultRAT()

vignette("raster-attribute-tables")

Examples

```
# see examples for `buildRAT()`
```

---

dump_open_datasets *Report open datasets*

Description

dump_open_datasets() dumps a list of all open datasets (shared or not) to the console. This function is primarily intended to assist in debugging "dataset leaks" and reference counting issues. The information reported includes the dataset name, referenced count, shared status, driver name, size, and band count. This a wrapper for GDALDumpOpenDatasets() with output to the console.

Usage

```
dump_open_datasets()
```

Value

Number of open datasets.

Examples

```
elev_file <- system.file("extdata/storml_elev.tif", package="gdalraster")
ds <- new(GDALRaster, elev_file)
dump_open_datasets()
ds2 <- new(GDALRaster, elev_file)
dump_open_datasets()
# open without using shared mode
ds3 <- new(GDALRaster, elev_file, read_only = TRUE,
           open_options = NULL, shared = FALSE)
dump_open_datasets()
ds$close()
```

```
dump_open_datasets()
ds2$close()
dump_open_datasets()
ds3$close()
dump_open_datasets()
```

`fillNodata` *Fill selected pixels by interpolation from surrounding areas*

Description

`fillNodata()` is a wrapper for `GDALFillNodata()` in the GDAL Algorithms API. This algorithm will interpolate values for all designated nodata pixels (pixels having an intrinsic nodata value, or marked by zero-valued pixels in the optional raster specified in `mask_file`). For each nodata pixel, a four direction conic search is done to find values to interpolate from (using inverse distance weighting). Once all values are interpolated, zero or more smoothing iterations (3x3 average filters on interpolated pixels) are applied to smooth out artifacts.

Usage

```
fillNodata(
  filename,
  band,
  mask_file = "",
  max_dist = 100,
  smooth_iterations = 0L,
  quiet = FALSE
)
```

Arguments

`filename`	Filename of input raster in which to fill nodata pixels.
`band`	Integer band number to modify in place.
`mask_file`	Optional filename of raster to use as a validity mask (band 1 is used, zero marks nodata pixels, non-zero marks valid pixels).
`max_dist`	Maximum distance (in pixels) that the algorithm will search out for values to interpolate (100 pixels by default).
`smooth_iterations`	The number of 3x3 average filter smoothing iterations to run after the interpolation to dampen artifacts (0 by default).
`quiet`	Logical scalar. If `TRUE`, a progress bar will not be displayed. Defaults to `FALSE`.

Value

Logical indicating success (invisible `TRUE`). An error is raised if the operation fails.

Note

The input raster will be modified in place. It should not be open in a GDALRaster object while processing with fillNodata().

Examples

```
## fill nodata edge pixels
f <- system.file("extdata/storml_elev_orig.tif", package="gdalraster")

## get count of nodata
tbl <- buildRAT(f)
head(tbl)
tbl[is.na(tbl$VALUE),]

ds <- new(GDALRaster, f)
plot_raster(ds, legend = TRUE)
ds$close()

## make a copy that will be modified
mod_file <- file.path(tempdir(), "storml_elev_fill.tif")
file.copy(f,  mod_file)

fillNodata(mod_file, band = 1)

mod_tbl = buildRAT(mod_file)
head(mod_tbl)
mod_tbl[is.na(mod_tbl$VALUE),]

ds <- new(GDALRaster, mod_file)
plot_raster(ds, legend = TRUE)
ds$close()
```

---

footprint *Compute footprint of a raster*

Description

footprint() is a wrapper of the gdal_footprint command-line utility (see https://gdal.org/en/stable/programs/gdal_footprint.html). The function can be used to compute the footprint of a raster file, taking into account nodata values (or more generally the mask band attached to the raster bands), and generating polygons/multipolygons corresponding to areas where pixels are valid, and write to an output vector file. Refer to the GDAL documentation at the URL above for a list of command-line arguments that can be passed in cl_arg. Requires GDAL >= 3.8.

Usage

```
footprint(src_filename, dst_filename, cl_arg = NULL)
```

Arguments

src_filename	Character string. Filename of the source raster.
dst_filename	Character string. Filename of the destination vector. If the file and the output layer exist, the new footprint is appended to them, unless the -overwrite command-line argument is used.
cl_arg	Optional character vector of command-line arguments for gdal_footprint.

Details

Post-vectorization geometric operations are applied in the following order:

- optional splitting (-split_polys)
- optional densification (-densify)
- optional reprojection (-t_srs)
- optional filtering by minimum ring area (-min_ring_area)
- optional application of convex hull (-convex_hull)
- optional simplification (-simplify)
- limitation of number of points (-max_points)

Value

Logical indicating success (invisible TRUE). An error is raised if the operation fails.

See Also

polygonize()

Examples

```
evt_file <- system.file("extdata/storml_evt.tif", package="gdalraster")
out_file <- file.path(tempdir(), "storml.geojson")

# Requires GDAL >= 3.8
if (gdal_version_num() >= gdal_compute_version(3, 8, 0)) {
  # command-line arguments for gdal_footprint
  args <- c("-t_srs", "EPSG:4326")
  footprint(evt_file, out_file, args)

}
```

GDALAlg-class *Class encapsulating a GDAL CLI algorithm*

Description

GDALAlg provides bindings to GDALAlgorithm and related classes that implement the gdal command line interface (CLI) in the GDAL API. An object of class GDALAlg represents an instance of a CLI algorithm with methods to obtain algorithm information and argument information, run the algorithm, and access its output.

Requires GDAL >= 3.11.3.

Experimental (see the section Development Status below)

GDALAlg is a C++ class exposed directly to R (via RCPP_EXPOSED_CLASS). Fields and methods of the class are accessed using the $ operator. Arguments to class constructors and class methods must be given in the order documented (naming optional).

Arguments

cmd	A character string or character vector containing the path to the algorithm, e.g., "raster reproject" or c("raster", "reproject").
args	Either a character vector or a named list containing input arguments of the algorithm (see section Algorithm Argument Syntax below).

Value

An object of class GDALAlg, which contains a pointer to the algorithm instance. Class methods are described in Details, along with a set of writable fields for per-object settings.

Usage (see Details)

```
## Constructors
alg <- new(GDALAlg, cmd)
# or, with arguments
alg <- new(GDALAlg, cmd, args)

## Read/write fields (per-object settings)
alg$setVectorArgsFromObject
alg$outputLayerNameForOpen
alg$quiet

## Methods
alg$info()
alg$argInfo(arg_name)
alg$usage()
alg$usageAsJSON()

alg$setArg(arg_name, arg_value)
```

```
alg$parseCommandLineArgs()
alg$getExplicitlySetArgs()
alg$run()
alg$output()
alg$outputs()

alg$close()
alg$release()
```

Details

Constructors:

`new(GDALAlg, cmd)`
Instantiate an algorithm without specifying input arguments.

`new(GDALAlg, cmd, args)`
Instantiate an algorithm giving input arguments as a character vector or named list. See the section `Algorithm Argument Syntax` for details.

Read/write fields (per-object settings):

`$setVectorArgsFromObject`
A logical value, `TRUE` (the default) to set algorithm arguments automatically when the `"input"` argument or the `"like"` argument is an object of class `GDALVector`. Argument values specified explicitly will override the automatic setting (as long as they result in a parsable set of arguments). Automatically setting arguments from `GDALVector` input can be disabled by setting this field to `FALSE`. In that case, only the vector dataset would be passed to the algorithm, i.e., without automatically passing any layer specifications. When enabled and the `"input"` or `"like"` argument for an algorithm is given as a `GDALVector` object, corresponding arguments will be set automatically based on properties of the object (when the argument is available to the algorithm):

- `"input-format"`: set to the `GDALVector` object's driver short name
- `"input-layer"`: set to the `GDALVector` layer name if it is not a SQL layer
- `"sql"`: set to the SQL statement if the `GDALVector` layer is defined by one
- `"dialect"`: set to the SQL dialect if one is specified for a SQL layer
- `"like-layer"`: set to the `GDALVector` layer name if it is not a SQL layer
- `"like-sql"`: set to the SQL statement if the `GDALVector` layer is defined by one

`$outputLayerNameForOpen`
A character string specifying a layer name to open when obtaining algorithm output as an object of class `GDALVector`. See method `$output()` below. The default value is empty string (`""`) in which case the first layer by index is opened. Ignored if output is not a vector dataset.

`$quiet`
A logical value, `FALSE` by default. Set to `TRUE` to suppress progress reporting along with various messages and warnings.

Methods:

`$info()`
Returns a named list of algorithm properties with the following elements:

- `name`: character string, the algorithm name
- `full_path`: character string, the algorithm name as full path

- description: character string, the algorithm (short) description
- long_description: character string, the algorithm longer description
- URL: character string, the algorithm help URL
- has_subalgorithms: logical, TRUE if the algorithm has sub-algorithms
- subalgorithm_names: character vector of sub-algorithm names (may be empty)parseCommandLineArgsparseComma
- arg_names: character vector of available argument names

$argInfo(arg_name)
Returns a named list of properties for an algorithm argument given as a character string, with the following elements:

- name: character string, the name of the argument
- type: character string, the argument type as one of "BOOLEAN", "STRING", "INTEGER", "REAL", "DATASET", "STRING_LIST", "INTEGER_LIST", "REAL_LIST", "DATASET_LIST"
- description: character string, the argument description
- short_name: character string, the short name or empty string if there is none
- aliases: character vector of aliases (empty if none)
- meta_var: character string, the "meta-var" hint (by default, the meta-var value is the long name of the argument in upper case)
- category: character string, the argument category
- is_positional: logical, TRUE if the argument is a positional one
- is_required: logical, TRUE if the argument is required
- min_count: integer, the minimum number of values for the argument (only applies to list type of arguments)
- max_count: integer, the maximum number of values for the argument (only applies to list type of arguments)
- packed_values_allowed: logical, TRUE if, for list type of arguments, several comma-separated values may be specified (i.e., "--foo=bar,baz")
- repeated_arg_allowed: logical, TRUE if, for list type of arguments, the argument may be repeated (i.e., c("--foo=bar", "--foo=baz"))
- choices: character vector of allowed values for the argument (may be empty and only applies for argument types "STRING" and "STRING_LIST")
- is_explicitly_set: logical, TRUE if the argument value has been explicitly set
- has_default_value: logical, TRUE if the argument has a declared default value
- default_value: character or integer or numeric, the default value if the argument has one, otherwise NULL
- is_hidden_for_api: logical, TRUE if the argument is only for CLI usage (e.g., "–help") (GDAL >= 3.12 only, see is_only_for_cli)
- is_hidden_for_cli: logical, TRUE if the argument must not be mentioned in CLI usage (e.g., "output-value" for "gdal raster info", which is only meant when the algorithm is used from a non-CLI context such as programmatically from R)
- is_only_for_cli: logical, TRUE if the argument is only for CLI usage (e.g., "–help") (deprecated at GDAL 3.12, see is_hidden_for_api)
- is_input: logical, TRUE if the value of the argument is read-only during the execution of the algorithm

- is_output: logical, TRUE if (at least part of) the value of the argument is set during the execution of the algorithm
- dataset_type_flags: character vector containing strings "RASTER", "VECTOR", "MULTIDIM_RASTER", possibly with "UPDATE" (NULL if the argument is not a dataset type)
- dataset_input_flags: character vector indicating if a dataset argument supports specifying only the dataset name ("NAME"), only the dataset object ("OBJECT"), or both ("NAME", "OBJECT") when it is used as an input (NULL if the argument is not a dataset type)
- dataset_output_flags: character vector indicating if a dataset argument supports specifying only the dataset name ("NAME"), only the dataset object ("OBJECT"), or both ("NAME", "OBJECT") when it is used as an output (NULL if the argument is not a dataset type)
- mutual_exclusion_group: character string, the name of the mutual exclusion group to which this argument belongs

$usage()
Print a help message for the algorithm to the console. No return value.

$usageAsJSON()
Returns the usage of the algorithm as a JSON-serialized string.

$setArg(arg_name, arg_value)
Set the value of input algorithm argument arg_name, a character string containing the argument's "long" name or an alias. The type of the arg_value parameter must be compatible with the algorithm argument type. Objects of class GDALRaster or GDALVector may be passed for algorithm arguments that accept dataset object input. A list of GDALRaster or GDALVector objects may be given for algorithm arguments of type DATASET_LIST that accept object input. Generally, an input dataset can also be specified by name as a character string (DSN), or character vector of DSNs for a DATASET_LIST.

$parseCommandLineArgs()
Sets the value of arguments previously specified in the class constructor, and instantiates the actual algorithm that will be run (but without running it). Returns a logical value, TRUE indicating success or FALSE if an error occurs.

$getExplicitlySetArgs()
Returns a named list of arguments that have been set explicitly along with their values. For arguments with dataset object values, the value of the list element is a string of the form "<TYPE dataset object: DSN>" where TYPE is one of "raster", "vector" or "multidim raster" and DSN is the dataset name (filename, URL, etc.).

$run()
Executes the algorithm, first parsing arguments if $parseCommandLineArgs() has not already been called explicitly. Returns a logical value, TRUE indicating success or FALSE if an error occurs.

$output()
Returns the single output value of the algorithm, after it has been run. If there are multiple output values, this method will raise an error, and the $outputs() (plural) method should be called instead. The type of the return value corresponds to the type of the single output argument value (see method $argInfo() above). If the output argument has type "DATASET", an object of class GDALRaster will be returned if the dataset is raster, or an object of class GDALVector if the dataset is vector. In the latter case, by default the GDALVector object will be opened on the first layer by index, but a specific layer name may be specified by setting the value of the field $outputLayerNameForOpen before calling the $output() method (see above). Note that currently, if the output dataset is multidimensional raster, only the dataset name will be returned as a character string.

`$outputs()`
Returns the output value(s) of the algorithm as a named list, after it has been run. Most algorithms only return a single output, in which case the `$output()` method (singular) is preferable for easier use. The element names in the returned list are the names of the arguments that have outputs (with any dash characters replaced by underscore), and the values are the argument values which may include `GDALRaster` or `GDALVector` objects.

`$close()`
Completes any pending actions, and returns the final status as a logical value (`TRUE` if no errors occur during the underlying call to `GDALAlgorithmFinalize()`). This is typically useful for algorithms that generate an output dataset. It closes datasets and gets back potential error status resulting from that, e.g., if an error occurs during flushing to disk of the output dataset after successful `$run()` execution.

`$release()`
Release memory associated with the algorithm, potentially after attempting to finalize. No return value, called for side-effects.

Algorithm Argument Syntax

Arguments are given in R as a character vector or named list, but otherwise syntax basically matches the GDAL specification for arguments as they are given on the command line. Those specifications are listed here along with some amendments regarding the character vector and named list formats. Programmatic usage also allows passing and receiving datasets as objects (i.e., `GDALRaster` or `GDALVector`), in addition to dataset names (e.g., filename, URL, database connection string).

- Commands accept one or several positional arguments, typically for dataset names (or in R as `GDALRaster` or `GDALVector` datasets). The order is input(s) first, output last. Positional arguments can also be specified as named arguments, if preferred to avoid any ambiguity.
- Named arguments have:
  - at least one "long" name, preceded by two dash characters
  - optionally, auxiliary long names (i.e., aliases),
  - and optionally a one-letter short name, preceded by a single dash character, e.g., `-f`, `--of`, `--format`, `--output-format` `<OUTPUT-FORMAT>`
- Boolean arguments are specified by just specifying the argument name in character vector format. In R `list` format, the named element must be assigned a value of logical `TRUE`.
- Arguments that require a value are specified like:
  - `-f VALUE` for one-letter short names
  - `--format VALUE` or `--format=VALUE` for long names
  - in a named list, this might look like: `args$format <- VALUE`
- Some arguments can be multi-valued. Some of them require all values to be packed together and separated with comma. This is, e.g., the case of:
  `--bbox <BBOX> Clipping bounding box as xmin,ymin,xmax,ymax`
  e.g., `--bbox=2.1,49.1,2.9,49.9`
- Others accept each value to be preceded by a new mention of the argument name, e.g., `c("--co", "COMPRESS=LZW", "--co", "TILED=YES")`. For that one, if the value of the argument does not contain commas, the packed form is also accepted: `--co COMPRESS=LZW,TILED=YES`.

Note that repeated mentions of an argument are possible in the character vector format for argument input, whereas arguments given in named list format must use argument long names as the list element names, and the packed format for the values (which can be a character vector or numeric vector of values).

- Named arguments can be placed before or after positional arguments.

Development Status

The GDAL Command Line Interface Modernization was first introduced in the GDAL 3.11.0 release (2025-05-09). The GDAL project provides warning that the new CLI "is provisionally provided as an alternative interface to GDAL and OGR command line utilities. The project reserves the right to modify, rename, reorganize, and change the behavior until it is officially frozen via PSC vote in a future major GDAL release. The utility needs time to mature, benefit from incremental feedback, and explore enhancements without carrying the burden of full backward compatibility. Your usage of it should have no expectation of compatibility until that time." (https://gdal.org/en/latest/programs/#gdal-application)

Initial bindings to enable programmatic use of the CLI algorithms from R were added in **gdalraster** 2.2.0, and will evolve over future releases. *The bindings are considered experimental until the upstream API is declared stable*. Breaking changes in minor version releases are possible until then. Please use with those cautions in mind. Bug reports may be filed at: https://github.com/USDAForestService/gdalraster/issues.

See Also

gdal_alg(), gdal_commands(), gdal_run(), gdal_usage()

Using gdal CLI algorithms from R
https://usdaforestservice.github.io/gdalraster/articles/use-gdal-cli-from-r.html

Examples

```
f <- system.file("extdata/storml_elev.tif", package="gdalraster")

## raster info
gdal_usage("raster info")

# arguments given as character vector
args <- c("--format=text", "--no-md", f)
(alg <- new(GDALAlg, "raster info", args))

alg$run()

alg$output() |> writeLines()

alg$release()

## raster clip
gdal_usage("raster clip")

# arguments in list format
args <- list()
```

```
args$input <- f
f_clip <- file.path(tempdir(), "elev_clip.tif")
args$output <- f_clip
args$overwrite <- TRUE
args$bbox <- c(323776.1, 5102172.0,  327466.1, 5104782.0)

(alg <- new(GDALAlg, "raster clip", args))

alg$run()

# get output as a GDALRaster object
(ds_clip <- alg$output())

alg$release()

ds_clip$dim()
ds_clip$bbox()

## raster hillshade
gdal_usage("raster hillshade")

# input as a GDALRaster object and output to an in-memory raster
args <- list(input = ds_clip,
             output_format = "MEM",
             output = "")

(alg <- new(GDALAlg, "raster hillshade", args))

alg$run()

(ds_hillshade <- alg$output())

alg$release()

plot_raster(ds_hillshade)

ds_clip$close()
ds_hillshade$close()
deleteDataset(f_clip)

## vector convert shapefile to GeoPackage
gdal_usage("vector convert")

f_shp <- system.file("extdata/poly_multipoly.shp", package="gdalraster")
f_gpkg <- file.path(tempdir(), "polygons_test.gpkg")
args <- c("--input", f_shp, "--output", f_gpkg, "--overwrite")

(alg <- new(GDALAlg, "vector convert", args))

alg$run()

# output as a GDALVector object
(lyr <- alg$output())
```

```
alg$release()

lyr$info()

lyr$close()
deleteDataset(f_gpkg)

## raster reproject (usage and info)
# no arguments given, only for retrieving usage and properties via API
# cf. gdal_usage("raster reproject")

(alg <- new(GDALAlg, "raster reproject"))

# print usage to the console, no return value
alg$usage()

# or, get usage as a JSON string
# json <- alg$usageAsJSON()
# writeLines(json)

# list of algorithm properties
alg$info()

# list of properties for an algorithm argument
alg$argInfo("resampling")

alg$release()
```

GDALRaster-class *Class encapsulating a raster dataset and associated band objects*

Description

GDALRaster provides an interface for accessing a raster dataset via GDAL and calling methods on the underlying GDALDataset, GDALDriver and GDALRasterBand objects. See https://gdal.org/en/stable/api/index.html for details of the GDAL Raster API.

GDALRaster is a C++ class exposed directly to R (via RCPP_EXPOSED_CLASS). Fields and methods of the class are accessed using the $ operator. **Note that all arguments to class methods are required and must be given in the order documented.** Naming the arguments is optional but may be preferred for readability.

Arguments

filename	Character string containing the file name of a raster dataset to open, as full path or relative to the current working directory. In some cases, filename may not refer to a local file system, but instead contain format-specific information on how to access a dataset such as database connection string, URL, /vsiPREFIX/,

etc. (see GDAL raster format descriptions: https://gdal.org/en/stable/drivers/raster/index.html).

`read_only` Logical. `TRUE` to open the dataset read-only (the default), or `FALSE` to open with write access.

`open_options` Optional character vector of `NAME=VALUE` pairs specifying dataset open options.

`shared` Logical. `FALSE` to open the dataset without using shared mode. Default is `TRUE` (see Note).

`allowed_drivers` Optional character vector of driver short names that must be considered. By default, all known raster drivers are considered.

Value

An object of class `GDALRaster`, which contains a pointer to the opened dataset. Class methods that operate on the dataset are described in Details, along with a set of writable fields for per-object settings. Values may be assigned to the class fields as needed during the lifetime of the object (i.e., by regular `<-` or `=` assignment).

Usage (see Details)

```
## Constructors
# read-only by default:
ds <- new(GDALRaster, filename)
# for update access:
ds <- new(GDALRaster, filename, read_only = FALSE)
# to specify dataset open options:
ds <- new(GDALRaster, filename, read_only = TRUE|FALSE, open_options)
# to open without using shared mode:
new(GDALRaster, filename, read_only, open_options, shared = FALSE)
# to specify certain allowed driver(s):
new(GDALRaster, filename, read_only, open_options, shared, allowed_drivers)

## Read/write fields (per-object settings)
ds$infoOptions
ds$quiet
ds$readByteAsRaw

## Methods
ds$getFilename()
ds$setFilename(filename)
ds$open(read_only)
ds$isOpen()
ds$getFileList()

ds$info()
ds$infoAsJSON()

ds$getDriverShortName()
```

```
ds$getDriverLongName()

ds$getRasterXSize()
ds$getRasterYSize()
ds$getRasterCount()

ds$addBand(dataType, options)

ds$getGeoTransform()
ds$setGeoTransform(transform)

ds$getProjection()
ds$getProjectionRef()
ds$setProjection(projection)

ds$bbox()
ds$res()
ds$dim()

ds$apply_geotransform(col_row)
ds$get_pixel_line(xy)

ds$get_block_indexing(band)
ds$make_chunk_index(band, max_pixels)

ds$getDescription(band)
ds$setDescription(band, desc)
ds$getBlockSize(band)
ds$getActualBlockSize(band, xblockoff, yblockoff)
ds$getOverviewCount(band)
ds$buildOverviews(resampling, levels, bands)
ds$getDataTypeName(band)
ds$getNoDataValue(band)
ds$setNoDataValue(band, nodata_value)
ds$deleteNoDataValue(band)
ds$getMaskFlags(band)
ds$getMaskBand(band)
ds$getUnitType(band)
ds$setUnitType(band, unit_type)
ds$getScale(band)
ds$setScale(band, scale)
ds$getOffset(band)
ds$setOffset(band, offset)
ds$getRasterColorInterp(band)
ds$setRasterColorInterp(band, col_interp)

ds$getMinMax(band, approx_ok)
ds$getMinMaxLocation(band)
```

```
ds$getStatistics(band, approx_ok, force)
ds$clearStatistics()

ds$getHistogram(band, min, max, num_buckets, incl_out_of_range, approx_ok)
ds$getDefaultHistogram(band, force)

ds$getMetadata(band, domain)
ds$setMetadata(band, metadata, domain)
ds$getMetadataItem(band, mdi_name, domain)
ds$setMetadataItem(band, mdi_name, mdi_value, domain)
ds$getMetadataDomainList(band)

ds$read(band, xoff, yoff, xsize, ysize, out_xsize, out_ysize)
ds$readBlock(band, xblockoff, yblockoff)
ds$readChunk(band, chunk_def)

ds$write(band, xoff, yoff, xsize, ysize, rasterData)
ds$writeBlock(band, xblockoff, yblockoff, rasterData)
ds$writeChunk(band, chunk_def, rasterData)
ds$fillRaster(band, value, ivalue)

ds$getColorTable(band)
ds$getPaletteInterp(band)
ds$setColorTable(band, col_tbl, palette_interp)
ds$clearColorTable(band)

ds$getDefaultRAT(band)
ds$setDefaultRAT(band, df)

ds$flushCache()

ds$getChecksum(band, xoff, yoff, xsize, ysize)

ds$close()
```

Details

Constructors:

`new(GDALRaster, filename, read_only)`
Returns an object of class `GDALRaster`. The `read_only` argument defaults to `TRUE` if not specified.

`new(GDALRaster, filename, read_only, open_options)`
Alternate constructor for passing dataset `open_options`, a character vector of `NAME=VALUE` pairs. `read_only` is required for this form of the constructor, `TRUE` for read-only access, or `FALSE` to open with write access. Returns an object of class `GDALRaster`.

`new(GDALRaster, filename, read_only, open_options, shared)`
Alternate constructor for specifying the `shared` mode for dataset opening. The `shared` argument defaults to `TRUE` but can be set to `FALSE` with this constructor (see Note). All arguments are required with this form of the constructor, but `open_options` can be `NULL`. Returns an object of

class GDALRaster.

new(GDALRaster, filename, read_only, open_options, shared, allowed_drivers)
Alternate constructor for specifying the driver(s) allowed for dataset opening as a character vector of driver short names. All arguments are required with this form of the constructor, but open_options can be NULL. Returns an object of class GDALRaster.

Read/write fields:

$infoOptions
A character vector of command-line arguments to control the output of $info() and $infoAsJSON() (see below). Defaults to character(0). Can be set to a vector of strings specifying arguments to the gdalinfo command-line utility, e.g., c("-nomd", "-norat", "-noct"). Restore the default by setting to empty string ("") or character(0).

$quiet
A logical value, FALSE by default. This field can be set to TRUE which will suppress various messages as well as progress reporting for potentially long-running processes such as building overviews and computation of statistics and histograms.

$readByteAsRaw
A logical value, FALSE by default. This field can be set to TRUE which will affect the data type returned by the $read() method and the read_ds() convenience function. When the underlying band data type is Byte and readByteAsRaw is TRUE the output type will be raw rather than integer. See also the as_raw argument to read_ds() to control this in a non-persistent setting. If the underlying band data type is not Byte this setting has no effect.

Methods:

$getFilename()
Returns a character string containing the filename associated with this GDALRaster object (filename originally used to open the dataset). May be a regular filename, database connection string, URL, etc.

$setFilename(filename)
Sets the filename if the underlying dataset does not already have an associated filename. Explicitly setting the filename is an advanced setting that should only be used when the user has determined that it is needed. Writing certain virtual datasets to file is one potential use case (e.g., a dataset returned by autoCreateWarpedVRT()).

$open(read_only)
(Re-)opens the raster dataset on the existing filename. Use this method to open a dataset that has been closed using $close(). May be used to re-open a dataset with a different read/write access (read_only set to TRUE or FALSE). The method will first close an open dataset, so it is not required to call $close() explicitly in this case. No return value, called for side effects.

$isOpen()
Returns logical indicating whether the associated raster dataset is open.

$getFileList()
Returns a character vector of files believed to be part of this dataset. If it returns an empty string ("") it means there is believed to be no local file system files associated with the dataset (e.g., a virtual file system). The returned filenames will normally be relative or absolute paths depending on the path used to originally open the dataset.

$info()
Prints various information about the raster dataset to the console (no return value, called for that

side effect only). Equivalent to the output of the gdalinfo command-line utility (gdalinfo filename, if using the default infoOptions). See the field $infoOptions above for setting the arguments to gdalinfo.

$infoAsJSON()
Returns information about the raster dataset as a JSON-formatted string. Equivalent to the output of the gdalinfo command-line utility (gdalinfo -json filename, if using the default infoOptions). See the field $infoOptions above for setting the arguments to gdalinfo.

$getDriverShortName()
Returns the short name of the raster format driver.

$getDriverLongName()
Returns the long name of the raster format driver.

$getRasterXSize()
Returns the number of pixels along the x dimension.

$getRasterYSize()
Returns the number of pixels along the y dimension.

$getRasterCount()
Returns the number of raster bands on this dataset. For the methods described below that operate on individual bands, the band argument is the integer band number (1-based).

$addBand(dataType, options)
Adds a band to a dataset if the underlying format supports this action. Most formats do not, but "MEM" and "VRT" are notable exceptions that support adding bands. The added band will always be the last band. dataType is a character string containing the data type name (e.g., "Byte", "Int16", "UInt16", "Int32", "Float32", etc). The options argument is a character vector of NAME=VALUE option strings. Supported options are format specific. Note that the options argument is required but may be given as NULL. Returns logical TRUE on success or FALSE if a band could not be added.

$getGeoTransform()
Returns the affine transformation coefficients for transforming between pixel/line raster space (column/row) and projection coordinate space (geospatial x/y). The return value is a numeric vector of length six. See https://gdal.org/en/stable/tutorials/geotransforms_tut.html for details of the affine transformation. *With 1-based indexing in* R, the geotransform vector contains (in map units of the raster spatial reference system):

GT[1]	x-coordinate of upper-left corner of the upper-left pixel
GT[2]	x-component of pixel width
GT[3]	row rotation (zero for north-up raster)
GT[4]	y-coordinate of upper-left corner of the upper-left pixel
GT[5]	column rotation (zero for north-up raster)
GT[6]	y-component of pixel height (negative for north-up raster)

$setGeoTransform(transform)
Sets the affine transformation coefficients on this dataset. transform is a numeric vector of length six. Returns logical TRUE on success or FALSE if the geotransform could not be set.

$getProjection()
Returns the coordinate reference system of the raster as an OGC WKT format string. Equivalent to ds$getProjectionRef().

`$getProjectionRef()`
Returns the coordinate reference system of the raster as an OGC WKT format string. An empty string is returned when a projection definition is not available.

`$setProjection(projection)`
Sets the projection reference for this dataset. `projection` is a string in OGC WKT format. Returns logical `TRUE` on success or `FALSE` if the projection could not be set.

`$bbox()`
Returns a numeric vector of length four containing the bounding box (xmin, ymin, xmax, ymax).

`$res()`
Returns a numeric vector of length two containing the resolution (pixel width, pixel height as positive values) for a non-rotated raster. A warning is emitted and `NA` values returned if the raster has a rotated geotransform (see method `$getGeoTransform()` above).

`$dim()`
Returns an integer vector of length three containing the raster dimensions (xsize, ysize, number of bands). Equivalent to:

```
c(ds$getRasterXSize(), ds$getRasterYSize(), ds$getRasterCount())
```

`$apply_geotransform(col_row)`
Applies geotransform coefficients to raster coordinates in pixel/line space (column/row), converting into georeferenced (x/y) coordinates. `col_row` is a numeric matrix of raster col/row coordinates (or two-column data frame that will be coerced to numeric matrix). Returns a numeric matrix of geospatial x/y coordinates. See the stand-alone function of the same name (`apply_geotransform()`) for more info and examples.

`$get_pixel_line(xy)`
Converts geospatial coordinates to pixel/line (raster column/row numbers). `xy` is a numeric matrix of geospatial x,y coordinates in the same spatial reference system as the raster (or two-column data frame that will be coerced to numeric matrix). Returns an integer matrix of raster pixel/line. See the stand-alone function of the same name (`get_pixel_line()`) for more info and examples.

`$get_block_indexing(band)`
Helper method returning a numeric matrix with named columns: `xblockoff`, `yblockoff`, `xoff`, `yoff`, `xsize`, `ysize`, `xmin`, `xmax`, `ymin`, `ymax`. This method provides indexing values for the block layout of the given band number. See also the class methods: `$getBlockSize()`, `$getActualBlockSize()` and `$read()`. The returned matrix has number of rows equal to the number of blocks for the given band number, with blocks ordered left to right, top to bottom. All offsets are zero-based. The `xoff`/`yoff` values are pixel offsets to the start of a block. The `xsize`/`ysize` values give the actual block sizes accounting for potentially incomplete blocks along the right and bottom edges. The columns `xmin`, `xmax`, `ymin` and `ymax` give the extent of each block in geospatial coordinates.

`$make_chunk_index(band, max_pixels)`
Helper method returning a numeric matrix with named columns: `xchunkoff`, `ychunkoff`, `xoff`, `yoff`, `xsize`, `ysize`, `xmin`, `xmax`, `ymin`, `ymax`. This method generates indexing information (offsets, sizes and geospatial bounding boxes) for potentially multi-block chunks, defined on block boundaries for efficient I/O. The output of this method can be used with the `$readChunk()`/`$writeChunk()` methods to iterate I/O operations conveniently on user-defined chunk sizes. The chunks will contain at most `max_pixels` given as a numeric value optionally carrying the `bit64::integer64` class attribute (numeric values will be coerced to 64-bit integer internally by truncation). A value of `max_pixels = 0` (or any value less than raster block xsize * block ysize, see `$getBlockSize()` below), will return output equivalent to `$get_block_indexing()`. The returned matrix has number of rows equal to the number of chunks for the given band number, with chunks ordered left

to right, top to bottom. All offsets are zero-based. The xoff/yoff values are pixel offsets to the start of a chunk. The xsize/ysize values give the actual chunk sizes accounting for potentially incomplete chunks along the right and bottom edges. The columns xmin, xmax, ymin and ymax give the extent of each chunk in geospatial coordinates.

$getDescription(band)
Returns a string containing the description for band. An empty string is returned if no description is set for the band. Passing band = 0 will return the dataset-level description.

$setDescription(band, desc)
Sets a description for band. desc is the character string to set. No return value. (Passing band = 0 can be used to set the dataset-level description. Note that the dataset description is generally the filename that was used to open the dataset. It usually should not be changed by calling this method on an existing dataset.)

$getBlockSize(band)
Returns a numeric vector of length two (xsize, ysize) containing the "natural" block size of band. GDAL has a concept of the natural block size of rasters so that applications can organize data access efficiently for some file formats. The natural block size is the block size that is most efficient for accessing the format. For many formats this is simply a whole row in which case block xsize is the same as $getRasterXSize() and block ysize is 1. However, for tiled images block size will typically be the tile size. Note that the X and Y block sizes don't have to divide the image size evenly, meaning that right and bottom edge blocks may be incomplete.

$getActualBlockSize(band, xblockoff, yblockoff)
Returns a numeric vector of length two (xvalid, yvalid) containing the actual block size for a given block offset in band. Handles partial blocks at the edges of the raster and returns the true number of pixels. xblockoff is an integer value, the horizontal block offset for which to calculate the number of valid pixels, with zero indicating the left most block, 1 the next block, etc. yblockoff is likewise the vertical block offset, with zero indicating the top most block, 1 the next block, etc.

$getOverviewCount(band)
Returns the number of overview layers (a.k.a. pyramids) available for band.

$buildOverviews(resampling, levels, bands)
Build one or more raster overview images using the specified downsampling algorithm. resampling is a character string, one of AVERAGE, AVERAGE_MAGPHASE, RMS, BILINEAR, CUBIC, CUBICSPLINE, GAUSS, LANCZOS, MODE, NEAREST or NONE. levels is an integer vector giving the list of overview decimation factors to build (e.g., c(2, 4, 8)), or 0 to delete all overviews (at least for external overviews (.ovr) and GTiff internal overviews). bands is an integer vector giving a list of band numbers to build overviews for, or 0 to build for all bands. Note that for GTiff, overviews will be created internally if the dataset is open in update mode, while external overviews (.ovr) will be created if the dataset is open read-only. External overviews created in GTiff format may be compressed using the COMPRESS_OVERVIEW configuration option. All compression methods supported by the GTiff driver are available (e.g., set_config_option("COMPRESS_OVERVIEW", "LZW")). Since GDAL 3.6, COMPRESS_OVERVIEW is honored when creating internal overviews of GTiff files. The GDAL documentation for gdaladdo command-line utility describes additional configuration for overview building. See also set_config_option(). No return value, called for side effects.

$getDataTypeName(band)
Returns the name of the pixel data type for band. The possible data types are:

Unknown	Unknown or unspecified type

Byte	8-bit unsigned integer
Int8	8-bit signed integer (GDAL >= 3.7)
UInt16	16-bit unsigned integer
Int16	16-bit signed integer
UInt32	32-bit unsigned integer
Int32	32-bit signed integer
UInt64	64-bit unsigned integer (GDAL >= 3.5)
Int64	64-bit signed integer (GDAL >= 3.5)
Float32	32-bit floating point
Float64	64-bit floating point
CInt16	Complex Int16
CInt32	Complex Int32
CFloat32	Complex Float32
CFloat64	Complex Float64

Some raster formats including GeoTIFF ("GTiff") and Erdas Imagine .img ("HFA") support sub-byte data types. Rasters can be created with these data types by specifying the "NBITS=n" creation option where n=1...7 for GTiff or n=1/2/4 for HFA. In these cases, `$getDataTypeName()` reports the apparent type `"Byte"`. GTiff also supports n=9...15 (UInt16 type) and n=17...31 (UInt32 type), and n=16 is accepted for Float32 to generate half-precision floating point values.

`$getNoDataValue(band)`
Returns the nodata value for `band` if one exists. This is generally a special value defined to mark pixels that are not valid data. `NA` is returned if a nodata value is not defined for `band`. Not all raster formats support a designated nodata value.

`$setNoDataValue(band, nodata_value)`
Sets the nodata value for `band`. `nodata_value` is a numeric value to be defined as the nodata marker. Depending on the format, changing the nodata value may or may not have an effect on the pixel values of a raster that has just been created (often not). It is thus advised to call `$fillRaster()` explicitly if the intent is to initialize the raster to the nodata value. In any case, changing an existing nodata value, when one already exists on an initialized dataset, has no effect on the pixels whose values matched the previous nodata value. Returns logical `TRUE` on success or `FALSE` if the nodata value could not be set.

`$deleteNoDataValue(band)`
Removes the nodata value for `band`. This affects only the definition of the nodata value for raster formats that support one (does not modify pixel values). No return value. An error is raised if the nodata value cannot be removed.

`$getMaskFlags(band)`
Returns the status flags of the mask band associated with `band`. Masks are represented as Byte bands with a value of zero indicating nodata and non-zero values indicating valid data. Normally the value `255` will be used for valid data pixels. GDAL supports external (.msk) mask bands, and normal Byte alpha (transparency) band as mask (any value other than `0` to be treated as valid data). But masks may not be regular raster bands on the datasource, such as an implied mask from a band nodata value or the `ALL_VALID` mask. See RFC 15: Band Masks for more details (https://gdal.org/en/stable/development/rfc/rfc15_nodatabitmask.html.

Returns a named list of GDAL mask flags and their logical values, with the following definitions:

- `ALL_VALID`: There are no invalid pixels, all mask values will be `255`. When used this will normally be the only flag set.

- PER_DATASET: The mask band is shared between all bands on the dataset.
- ALPHA: The mask band is actually an alpha band and may have values other than 0 and 255.
- NODATA: Indicates the mask is actually being generated from nodata values (mutually exclusive of ALPHA).

$getMaskBand(band)
Returns the mask filename and band number associated with band. The return value is a named list with two elements. The MaskFile element gives the filename where the mask band is located, e.g., a file with the same name as the main dataset but suffixed with .msk in the case of a GDAL external mask file. MaskFile will be an empty string for the derived ALL_VALID and NODATA masks, which internally are freestanding bands not considered to be a part of a dataset. The MaskBand element gives the band number for a mask that is a regular alpha band in the main dataset or external mask file. BandNumber will be 0 for the ALL_VALID and NODATA masks.

$getUnitType(band)
Returns the name of the unit type of the pixel values for band (e.g., "m" or "ft"). An empty string ("") is returned if no units are available.

$setUnitType(band, unit_type)
Sets the name of the unit type of the pixel values for band. unit_type should be one of empty string "" (the default indicating it is unknown), "m" indicating meters, or "ft" indicating feet, though other nonstandard values are allowed. Returns logical TRUE on success or FALSE if the unit type could not be set.

$getScale(band)
Returns the pixel value scale (units value = (raw value * scale) + offset) for band. This value (in combination with the $getOffset() value) can be used to transform raw pixel values into the units returned by $getUnitType(). Returns NA if a scale value is not defined for this band.

$setScale(band, scale)
Sets the pixel value scale (units value = (raw value * scale) + offset) for band. Many raster formats do not implement this method. Returns logical TRUE on success or FALSE if the scale could not be set.

$getOffset(band)
Returns the pixel value offset (units value = (raw value * scale) + offset) for band. This value (in combination with the $getScale() value) can be used to transform raw pixel values into the units returned by $getUnitType(). Returns NA if an offset value is not defined for this band.

$setOffset(band, offset)
Sets the pixel value offset (units value = (raw value * scale) + offset) for band. Many raster formats do not implement this method. Returns logical TRUE on success or FALSE if the offset could not be set.

$getRasterColorInterp(band)
Returns a string describing the color interpretation for band. The color interpretation values and their meanings are:

Undefined	Undefined
Gray	Grayscale
Palette	Paletted (see associated color table)
Red	Red band of RGBA image
Green	Green band of RGBA image
Blue	Blue band of RGBA image
Alpha	Alpha (0=transparent, 255=opaque)

Hue	Hue band of HLS image
Saturation	Saturation band of HLS image
Lightness	Lightness band of HLS image
Cyan	Cyan band of CMYK image
Magenta	Magenta band of CMYK image
Yellow	Yellow band of CMYK image
Black	Black band of CMYK image
YCbCr_Y	Y Luminance
YCbCr_Cb	Cb Chroma
YCbCr_Cr	Cr Chroma

`$setRasterColorInterp(band, col_interp)`
Sets the color interpretation for `band`. See above for the list of valid values for `col_interp` (passed as a string).

`$getMinMax(band, approx_ok)`
Returns a numeric vector of length two containing the min/max values for `band`. If `approx_ok` is `TRUE` and the raster format knows these values intrinsically then those values will be returned. If that doesn't work, a subsample of blocks will be read to get an approximate min/max. If the band has a nodata value it will be excluded from the minimum and maximum. If `approx_ok` is `FALSE`, then all pixels will be read and used to compute an exact range.

`$getMinMaxLocation(band)`
Computes the min/max values for a band, and their locations. Pixels with value matching the nodata value or masked by the mask band are ignored. If the minimum or maximum value is hit in several locations, it is not specified which location will be returned. The locations are returned as column/row number (0-based), geospatial x/y in the coordinate system of the raster, and WGS84 longitude/latitude. Returns a numeric vector with the following named values: `min`, `min_col`, `min_row`, `min_geo_x`, `min_geo_y`, `min_wgs84_lon`, `min_wgs84_lat`, `max`, `max_col`, `max_row`, `max_geo_x`, `max_geo_y`, `max_wgs84_lon`, `max_wgs84_lat`. This method wraps `GDALComputeRasterMinMaxLocation()` in the GDAL C API requiring GDAL >= 3.11.

`$getStatistics(band, approx_ok, force)`
Returns a numeric vector of length four containing the minimum, maximum, mean and standard deviation of pixel values in `band` (excluding nodata pixels). Some raster formats will cache statistics allowing fast retrieval after the first request.

`approx_ok`:

- `TRUE`: Approximate statistics are sufficient, in which case overviews or a subset of raster tiles may be used in computing the statistics.
- `FALSE`: All pixels will be read and used to compute statistics (if computation is forced).

`force`:

- `TRUE`: The raster will be scanned to compute statistics. Once computed, statistics will generally be "set" back on the raster band if the format supports caching statistics. (Note: `ComputeStatistics()` in the GDAL API is called automatically here. This is a change in the behavior of `GetStatistics()` in the API, to a definitive `force`.)
- `FALSE`: Results will only be returned if it can be done quickly (i.e., without scanning the raster, typically by using pre-existing STATISTICS_xxx metadata items). `NA`s will be returned if statistics cannot be obtained quickly.

`$clearStatistics()`
Clear statistics. Only implemented for now in PAM supported datasets (Persistable Auxiliary Metadata via .aux.xml file). GDAL >= 3.2.

`$getHistogram(band, min, max, num_buckets, incl_out_of_range, approx_ok)`
Computes raster histogram for `band`. `min` is the lower bound of the histogram. `max` is the upper bound of the histogram. `num_buckets` is the number of buckets to use (bucket size is `(max - min) / num_buckets`). `incl_out_of_range` is a logical scalar: if `TRUE` values below the histogram range will be mapped into the first bucket and values above will be mapped into the last bucket, if `FALSE` out of range values are discarded. `approx_ok` is a logical scalar: `TRUE` if an approximate histogram is OK (generally faster), or `FALSE` for an exactly computed histogram. Returns the histogram as a numeric vector of length `num_buckets`.

`$getDefaultHistogram(band, force)`
Returns a default raster histogram for `band`. In the GDAL API, this method is overridden by derived classes (such as GDALPamRasterBand, VRTDataset, HFADataset...) that may be able to fetch efficiently an already stored histogram. `force` is a logical scalar: `TRUE` to force the computation of a default histogram; or if `FALSE` and no default histogram is available, a warning is emitted and the returned list has a 0-length histogram vector. Returns a list of length four containing named elements `min` (lower bound), `max` (upper bound), `num_buckets` (number of buckets), and `histogram` (a numeric vector of length `num_buckets`).

`$getMetadata(band, domain)`
Returns a character vector of all metadata `NAME=VALUE` pairs that exist in the specified `domain`, or empty string (`""`) if there are no metadata items in `domain` (metadata in the context of the GDAL Raster Data Model https://gdal.org/en/stable/user/raster_data_model.html). Set `band = 0` to retrieve dataset-level metadata, or to an integer band number to retrieve band-level metadata. Set `domain = ""` (empty string) to retrieve metadata in the default domain.

`$setMetadata(band, metadata, domain)`
Sets metadata in the specified `domain`. The `metadata` argument is given as a character vector of `NAME=VALUE` pairs. Pass `band = 0` to set dataset-level metadata, or pass an integer band number to set band-level metadata. Use `domain = ""` (empty string) to set an item in the default domain. Returns logical `TRUE` on success or `FALSE` if metadata could not be set.

`$getMetadataItem(band, mdi_name, domain)`
Returns the value of a specific metadata item named `mdi_name` in the specified `domain`, or empty string (`""`) if no matching item is found. Set `band = 0` to retrieve dataset-level metadata, or to an integer band number to retrieve band-level metadata. Set `domain = ""` (empty string) to retrieve an item in the default domain.

`$setMetadataItem(band, mdi_name, mdi_value, domain)`
Sets the value (`mdi_value`) of a specific metadata item named `mdi_name` in the specified `domain`. Pass `band = 0` to set dataset-level metadata, or pass an integer band number to set band-level metadata. Use `domain = ""` (empty string) to set an item in the default domain. Returns logical `TRUE` on success or `FALSE` if metadata could not be set.

`$getMetadataDomainList(band)`
Returns a character vector of metadata domains or empty string (`""`). Set `band = 0` to retrieve dataset-level domains, or to an integer band number to retrieve band-level domains.

`$read(band, xoff, yoff, xsize, ysize, out_xsize, out_ysize)`
Reads a region of raster data from `band`. The method takes care of pixel decimation / replication if the output size (`out_xsize * out_ysize`) is different than the size of the region being accessed (`xsize * ysize`). `xoff` is the pixel (column) offset to the top left corner of the region of the band to be accessed (zero to start from the left side). `yoff` is the line (row) offset to the top left corner

of the region of the band to be accessed (zero to start from the top). *Note that raster row/column offsets use 0-based indexing.* xsize is the width in pixels of the region to be accessed. ysize is the height in pixels of the region to be accessed. out_xsize is the width of the output array into which the desired region will be read (typically the same value as xsize). out_ysize is the height of the output array into which the desired region will be read (typically the same value as ysize). Returns a numeric or complex vector containing the values that were read. It is organized in left to right, top to bottom pixel order. NA will be returned in place of the nodata value if the raster dataset has a nodata value defined for this band. Data are read as R integer type when possible for the raster data type (Byte, Int8, Int16, UInt16, Int32), otherwise as type double (UInt32, Float32, Float64). No rescaling of the data is performed (see $getScale() and $getOffset() above). An error is raised if the read operation fails. See also the setting $readByteAsRaw above.

$readBlock(band, xblockoff, yblockoff)

Reads a block of raster data, without resampling. See the class methods $getBlockSize() and $getActualBlockSize() above for a description of raster blocks. This is a convenience method to read by block offsets. Returns a vector of pixel values with length equal to the actual block xsize * ysize, otherwise as described above for $read().

$readChunk(band, chunk_def)

Reads a potentially multi-block chunk of raster data, without resampling. This is a convenience method that can be used with the output of $make_chunk_index() (see above). The matrix of chunk offsets and sizes returned by $make_chunk_index() (or $get_block_indexing() to operate on blocks) can be used to iterate I/O over the raster in user-defined chunk sizes, i.e., a row of that matrix can be passed for the chunk_def argument. chunk_def can also be given as an numeric vector of length 4 containing the 0-based pixel offsets and pixel sizes for a chunk (xoff, yoff, xsize, ysize). Returns a vector of pixel values with length equal to the chunk xsize * ysize, otherwise as described above for $read().

$write(band, xoff, yoff, xsize, ysize, rasterData)

Writes a region of raster data to band. xoff is the pixel (column) offset to the top left corner of the region of the band to be accessed (zero to start from the left side). yoff is the line (row) offset to the top left corner of the region of the band to be accessed (zero to start from the top). *Note that raster row/column offsets use 0-based indexing.* xsize is the width in pixels of the region to write. ysize is the height in pixels of the region to write. rasterData is a numeric or complex vector containing values to write. It is organized in left to right, top to bottom pixel order. NA in rasterData should be replaced with a suitable nodata value prior to writing (see $getNoDataValue() and $setNoDataValue() above). An error is raised if the operation fails (no return value).

$writeBlock(band, xblockoff, yblockoff, rasterData)

Writes a block of raster data. See the class methods $getBlockSize() and $getActualBlockSize() above for a description of raster blocks. This is a convenience method to write by block offsets.. Otherwise, this method operates like $write(). The length of rasterData must be the same as the actual xsize * ysize of the destination block.

$writeChunk(band, chunk_def, rasterData)

Writes a potentially multi-block chunk of raster data. This is a convenience method that can be used to iterate I/O over the raster in user-defined chunk sizes, see method $readChunk() above. Otherwise, this method operates like $write(). The length of rasterData must be the same as xsize * ysize of the destination chunk.

$fillRaster(band, value, ivalue)

Fills band with a constant value. GDAL makes no guarantees about what values the pixels in newly created files are set to, so this method can be used to clear a band to a specified "default"

value. The fill `value` is passed as `numeric`, but this will be converted to the underlying raster data type before writing to the file. The `ivalue` argument allows setting the imaginary component of a complex value. Note that `ivalue` is a required argument but can be set to `0` for real data types. No return value. An error is raised if the operation fails.

`$getColorTable(band)`
Returns the color table associated with `band`, or `NULL` if there is no associated color table. The color table is returned as an integer matrix with five columns. To associate a color with a raster pixel, the pixel value is used as a subscript into the color table. This means that the colors are always applied starting at zero and ascending (see GDAL Color Table). Column 1 contains the pixel values. Interpretation of columns 2:5 depends on the value of `$getPaletteInterp()` (see below). For "RGB", columns 2:5 contain red, green, blue, alpha as 0-255 integer values.

`$getPaletteInterp(band)`
If `band` has an associated color table, this method returns a character string with the palette interpretation for columns 2:5 of the table. An empty string (`""`) is returned if `band` does not have an associated color table. The palette interpretation values and their meanings are:

- Gray: column 2 contains grayscale values (columns 3:5 unused)
- RGB: columns 2:5 contain red, green, blue, alpha
- CMYK: columns 2:5 contain cyan, magenta, yellow, black
- HLS: columns 2:4 contain hue, lightness, saturation (column 5 unused)

`$setColorTable(band, col_tbl, palette_interp)`
Sets the raster color table for `band`. `col_tbl` is an integer matrix or data frame with either four or five columns (see `$getColorTable()` above). Column 1 contains the pixel values. Valid values are integers 0 and larger (note that GTiff format supports color tables only for Byte and UInt16 bands). Negative values will be skipped with a warning emitted. Interpretation of columns 2:5 depends on the value of `$getPaletteInterp()` (see above). For RGB, columns 2:4 contain red, green, blue as 0-255 integer values, and an optional column 5 contains alpha transparency values (defaults to 255 opaque). `palette_interp` is a string, one of `Gray`, `RGB`, `CMYK` or `HLS` (see `$getPaletteInterp()` above). Returns logical `TRUE` on success or `FALSE` if the color table could not be set.

`$clearColorTable(band)`
Clears the raster color table for `band`. Returns logical `TRUE` on success or `FALSE` if the color table could not be cleared, e.g., if this action is not supported by the driver.

`$getDefaultRAT(band)`
Returns the Raster Attribute Table for `band` as a data frame, or `NULL` if there is no associated Raster Attribute Table. See the stand-alone function `buildRAT()` for details of the Raster Attribute Table format.

`$setDefaultRAT(band, df)`
Sets a default Raster Attribute Table for `band` from data frame `df`. The input data frame will be checked for attribute `"GDALRATTableType"` which can have values of `"thematic"` or `"athematic"` (for continuous data). Columns of the data frame will be checked for attribute `"GFU"` (for "GDAL field usage"). If the `"GFU"` attribute is missing, a value of `"Generic"` will be used (corresponding to `GFU_Generic` in the GDAL API, for general purpose field). Columns with other, specific field usage values should generally be present in `df`, such as fields containing the set of unique (discrete) pixel values (GFU `"MinMax"`), pixel counts (GFU `"PixelCount"`), class names (GFU `"Name"`), color values (GFUs `"Red"`, `"Green"`, `"Blue"`), etc. The data frame will also be checked for attributes `"Row0Min"` and `"BinSize"` which can have numeric values that describe linear binning. See the stand-alone function `buildRAT()` for details of the GDAL Raster Attribute Table

format and its representation as data frame.

`$flushCache()`
Flush all write cached data to disk. Any raster data written via GDAL calls, but buffered internally will be written to disk. Using this method does not preclude calling `$close()` to properly close the dataset and ensure that important data not addressed by `$flushCache()` is written in the file (see also `$open()` above). No return value, called for side effects.

`$getChecksum(band, xoff, yoff, xsize, ysize)`
Returns a 16-bit integer (0-65535) checksum from a region of raster data on `band`. Floating point data are converted to 32-bit integer so decimal portions of such raster data will not affect the checksum. Real and imaginary components of complex bands influence the result. `xoff` is the pixel (column) offset of the window to read. `yoff` is the line (row) offset of the window to read. *Raster row/column offsets use 0-based indexing.* `xsize` is the width in pixels of the window to read. `ysize` is the height in pixels of the window to read.

`$close()`
Closes the GDAL dataset (no return value, called for side effects). Calling `$close()` results in proper cleanup, and flushing of any pending writes. Forgetting to close a dataset opened in update mode on some formats such as GTiff could result in being unable to open it afterwards. The `GDALRaster` object is still available after calling `$close()`. The dataset can be re-opened on the existing `filename` with `$open(read_only=TRUE)` or `$open(read_only=FALSE)`.

Note

If a dataset object is opened with update access (`read_only = FALSE`), it is not recommended to open a new dataset on the same underlying `filename`.

Datasets are opened in shared mode by default. This allows the sharing of `GDALDataset` handles for a dataset with other callers that open shared on the same `filename`, if the dataset is opened from the same thread. Functions in `gdalraster` that do processing will open input datasets in shared mode. This provides potential efficiency for cases when an object of class `GDALRaster` is already open in read-only mode on the same `filename` (avoids overhead associated with initial dataset opening by using the existing handle, and potentially makes use of existing data in the GDAL block cache). Opening in shared mode can be disabled by specifying the optional `shared` parameter in the class constructor.

The `$read()` method will perform automatic resampling if the specified output size (`out_xsize * out_ysize`) is different than the size of the region being read (`xsize * ysize`). In that case, the `GDAL_RASTERIO_RESAMPLING` configuration option could also be set to override the default resampling to one of `BILINEAR`, `CUBIC`, `CUBICSPLINE`, `LANCZOS`, `AVERAGE` or `MODE` (see `set_config_option()`).

See Also

Package overview in `help("gdalraster-package")`

`vignette("raster-api-tutorial")`

`read_ds()` is a convenience wrapper for `GDALRaster$read()`

Examples

```
lcp_file <- system.file("extdata/storm_lake.lcp", package="gdalraster")
ds <- new(GDALRaster, lcp_file)
ds
```

```
## print information about the dataset to the console
ds$info()

## retrieve the raster format name
ds$getDriverShortName()
ds$getDriverLongName()

## retrieve a list of files composing the dataset
ds$getFileList()

## retrieve dataset parameters
ds$getRasterXSize()
ds$getRasterYSize()
ds$getGeoTransform()
ds$getProjection()
ds$getRasterCount()
ds$bbox()
ds$res()
ds$dim()

## retrieve some band-level parameters
ds$getDescription(band = 1)
ds$getBlockSize(band = 1)
ds$getOverviewCount(band = 1)
ds$getDataTypeName(band = 1)
# LCP format does not support an intrinsic nodata value so this returns NA:
ds$getNoDataValue(band = 1)

## LCP driver reports several dataset- and band-level metadata
## (format description at https://gdal.org/en/stable/drivers/raster/lcp.html)
## set band = 0 to retrieve dataset-level metadata
## set domain = "" (empty string) for the default metadata domain
ds$getMetadata(band = 0, domain = "")

## retrieve metadata for a band as a vector of name=value pairs
ds$getMetadata(band = 4, domain = "")

## retrieve the value of a specific metadata item
ds$getMetadataItem(band = 2, mdi_name = "SLOPE_UNIT_NAME", domain = "")

## read one row of pixel values from band 1 (elevation)
## raster row/column index are 0-based
## the upper left corner is the origin
## read the tenth row:
ncols <- ds$getRasterXSize()
rowdata <- ds$read(band = 1, xoff = 0, yoff = 9,
                   xsize = ncols, ysize = 1,
                   out_xsize = ncols, out_ysize = 1)
head(rowdata)

ds$close()
```

```
## create a new raster using lcp_file as a template
new_file <- file.path(tempdir(), "storml_newdata.tif")
rasterFromRaster(srcfile = lcp_file,
                 dstfile = new_file,
                 nbands = 1,
                 dtName = "Byte",
                 init = -9999)

ds_new <- new(GDALRaster, new_file, read_only = FALSE)

## write random values to all pixels
set.seed(42)
ncols <- ds_new$getRasterXSize()
nrows <- ds_new$getRasterYSize()
for (row in 0:(nrows - 1)) {
    rowdata <- round(runif(ncols, 0, 100))
    ds_new$write(band = 1,
                 xoff = 0,
                 yoff = row,
                 xsize = ncols,
                 ysize = 1,
                 rowdata)
}

## re-open in read-only mode when done writing
## this will ensure flushing of any pending writes (implicit $close)
ds_new$open(read_only = TRUE)

## getStatistics returns min, max, mean, sd, and sets stats in the metadata
ds_new$getStatistics(band = 1, approx_ok = FALSE, force = TRUE)
ds_new$getMetadataItem(band = 1, "STATISTICS_MEAN", "")

## close the dataset for proper cleanup
ds_new$close()


## using a GDAL Virtual File System handler '/vsicurl/'
## see: https://gdal.org/en/stable/user/virtual_file_systems.html
url <- "/vsicurl/https://raw.githubusercontent.com/"
url <- paste0(url, "usdaforestservice/gdalraster/main/sample-data/")
url <- paste0(url, "lf_elev_220_mt_hood_utm.tif")

set_config_option("GDAL_HTTP_CONNECTTIMEOUT", "20")
set_config_option("GDAL_HTTP_TIMEOUT", "20")
if (http_enabled() && vsi_stat(url)) {
  ds <- new(GDALRaster, url)
  plot_raster(ds, legend = TRUE, main = "Mount Hood elevation (m)")
  ds$close()
}
set_config_option("GDAL_HTTP_CONNECTTIMEOUT", "")
set_config_option("GDAL_HTTP_TIMEOUT", "")
```

GDALVector-class *Class encapsulating a vector layer in a GDAL dataset*

Description

GDALVector provides an interface for accessing a vector layer in a GDAL dataset and calling methods on the underlying OGRLayer object. An object of class GDALVector persists an open connection to the dataset, and exposes methods to:

- retrieve layer information
- set attribute and spatial filters
- set a list of ignored or selected columns
- traverse and read feature data by traditional row-based cursor, including an analog of DBI::dbFetch()
- read via column-oriented Arrow Array stream
- write new features in a layer
- edit/overwrite existing features
- upsert
- delete features
- perform data manipulation within transactions.

GDALVector is a C++ class exposed directly to R (via RCPP_EXPOSED_CLASS). Fields and methods of the class are accessed using the $ operator. **Note that all arguments to class methods are required and must be given in the order documented.** Most GDALVector methods take zero or one argument, so this is usually not an issue. Class constructors are the main exception. Naming the arguments is optional but may be preferred for readability.

Arguments

dsn	Character string containing the data source name (DSN), usually a filename or database connection string.
layer	Character string containing the name of a layer within the data source. May also be given as an SQL SELECT statement to be executed against the data source, defining a layer as the result set.
read_only	Logical scalar. TRUE to open the layer read-only (the default), or FALSE to open with write access.
open_options	Optional character vector of NAME=VALUE pairs specifying dataset open options.
spatial_filter	Optional character string containing a geometry in Well Known Text (WKT) format which represents a spatial filter.
dialect	Optional character string to control the statement dialect when SQL is used to define the layer. By default, the OGR SQL engine will be used, except for RDBMS drivers that will use their dedicated SQL engine, unless "OGRSQL" is explicitly passed as the dialect. The "SQLITE" dialect can also be used.

Value

An object of class GDALVector, which contains pointers to the opened layer and the GDAL dataset that owns it. Class methods that operate on the layer are described in Details, along with a set of writable fields for per-object settings. Values may be assigned to the class fields as needed during the lifetime of the object (i.e., by regular <- or = assignment).

Usage (see Details)

```
## Constructors
# for single-layer file formats such as shapefile
lyr <- new(GDALVector, dsn)
# specifying the layer name, or SQL statement defining the layer
lyr <- new(GDALVector, dsn, layer)
# for update access
lyr <- new(GDALVector, dsn, layer, read_only = FALSE)
# using dataset open options
lyr <- new(GDALVector, dsn, layer, read_only, open_options)
# setting a spatial filter and/or specifying the SQL dialect
lyr <- new(GDALVector, dsn, layer, read_only, open_options, spatial_filter, dialect)

## Read/write fields (per-object settings)
lyr$defaultGeomColName
lyr$returnGeomAs
lyr$promoteToMulti
lyr$convertToLinear
lyr$wkbByteOrder
lyr$arrowStreamOptions
lyr$quiet
lyr$transactionsForce

## Methods
lyr$open(read_only)
lyr$isOpen()
lyr$getDsn()
lyr$getFileList()
lyr$info()
lyr$getDriverShortName()
lyr$getDriverLongName()

lyr$getName()
lyr$getFieldNames()
lyr$testCapability()
lyr$getFIDColumn()
lyr$getGeomType()
lyr$getGeometryColumn()
lyr$getSpatialRef()
lyr$bbox()
lyr$getLayerDefn()
```

```
lyr$getFieldDomain(domain_name)

lyr$setAttributeFilter(query)
lyr$getAttributeFilter()
lyr$setIgnoredFields(fields)
lyr$setSelectedFields(fields)
lyr$getIgnoredFields()

lyr$setSpatialFilter(wkt)
lyr$setSpatialFilterRect(bbox)
lyr$getSpatialFilter()
lyr$clearSpatialFilter()

lyr$getFeatureCount()
lyr$getNextFeature()
lyr$setNextByIndex(i)
lyr$getFeature(fid)
lyr$resetReading()
lyr$fetch(n)

lyr$getArrowStream()
lyr$releaseArrowStream()

lyr$setFeature(feature)
lyr$createFeature(feature)
lyr$batchCreateFeature(feature_set)
lyr$upsertFeature(feature)
lyr$getLastWriteFID()
lyr$deleteFeature(fid)
lyr$syncToDisk()

lyr$startTransaction()
lyr$commitTransaction()
lyr$rollbackTransaction()

lyr$getMetadata()
lyr$setMetadata(metadata)
lyr$getMetadataItem(mdi_name)

lyr$close()
```

Details

Constructors:

`new(GDALVector, dsn)`
The first layer by index is assumed if the `layer` argument is omitted, so this form of the constructor might be used for single-layer formats like shapefile.

`new(GDALVector, dsn, layer)`
Constructor specifying the name of a layer to open. The `layer` argument may also be given as an SQL SELECT statement to define a layer as the result set.

`new(GDALVector, dsn, layer, read_only)`
Constructor specifying read/write access (`read_only = TRUE|FALSE`). The `layer` argument is required in this form of the constructor, but may be given as empty string (`""`), in which case the first layer by index will be assumed.

`new(GDALVector, dsn, layer, read_only, open_options)`
Constructor specifying dataset open options as a character vector of `NAME=VALUE` pairs.

`new(GDALVector, dsn, layer, read_only, open_options, spatial_filter, dialect))`
Constructor to specify a spatial filter and/or SQL dialect. All arguments are required in this form of the constructor, but `open_options` may be `NULL`, and `spatial_filter` or `dialect` may be an empty string (`""`).

Read/write fields:

`$defaultGeomColName`
Character string specifying a name to use for returned columns when the geometry column name in the source layer is empty, like with shapefiles etc. Defaults to `"geom"`.

`$returnGeomAs`
Character string specifying the return format of feature geometries. Must be one of `WKB` (the default), `WKB_ISO`, `WKT`, `WKT_ISO`, `BBOX`, or `NONE`. Using `WKB`/`WKT` exports as 99-402 extended dimension (Z) types for Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon and GeometryCollection. For other geometry types, it is equivalent to using `WKB_ISO`/`WKT_ISO` (see https://libgeos.org/specifications/wkb/). Using `BBOX` exports as a list of numeric vectors, each of length 4 with values `xmin, ymin, xmax, ymax`. If an empty geometry is encountered these values will be `NA_real_` in the corresponding location. Using `NONE` will result in no geometry value being present in the feature returned.

`$promoteToMulti`
A logical value specifying whether to automatically promote geometries from Polygon to MultiPolygon, Point to MultiPoint, or LineString to MultiLineString during read operations (i.e., with methods `$getFeature()`, `$getNextFeature()`, `$fetch()`). Defaults to `FALSE`. Setting to `TRUE` may be useful when reading from layers such as shapefiles that mix Polygons and MultiPolygons.

`$convertToLinear`
A logical value specifying whether to convert non-linear geometry types into linear geometry types by approximating them (i.e., during read operations with methods `$getFeature()`, `$getNextFeature()`, `$fetch()`). Defaults to `FALSE`. If set to `TRUE`, handled conversions are:

- wkbCurvePolygon -> wkbPolygon
- wkbCircularString -> wkbLineString
- wkbCompoundCurve -> wkbLineString
- wkbMultiSurface -> wkbMultiPolygon
- wkbMultiCurve -> wkbMultiLineString

`$wkbByteOrder`
Character string specifying the byte order for WKB geometries. Must be either `LSB` (Least Significant Byte first, the default) or `MSB` (Most Significant Byte first).

`$arrowStreamOptions`
Character vector of `"NAME=VALUE"` pairs giving options used by the `$getArrowStream()` method

(see below). The available options may be driver and GDAL version specific. Options available as of GDAL 3.8 are listed below. For more information about options for Arrow stream, see the GDAL API documentation for OGR_L_GetArrowStream().

- INCLUDE_FID=YES/NO. Defaults to YES.
- MAX_FEATURES_IN_BATCH=integer. Maximum number of features to retrieve in an ArrowArray batch. Defaults to 65536.
- TIMEZONE=unknown/UTC/(+|:)HH:MM or any other value supported by Arrow (GDAL >= 3.8).
- GEOMETRY_METADATA_ENCODING=OGC/GEOARROW (GDAL >= 3.8). The GDAL default is OGC if not specified.
- GEOMETRY_ENCODING=WKB (Arrow/Parquet drivers). To force a fallback to the generic implementation when the native geometry encoding is not WKB. Otherwise the geometry will be returned with its native Arrow encoding (possibly using GeoArrow encoding).

`$quiet`
A logical value, `FALSE` by default. Set to `TRUE` to suppress various messages and warnings.

`$transactionsForce`
A logical value, `FALSE` by default. Affects the behavior of attempted transactions on the layer (see the `$startTransaction()` method below). By default, only "efficient" transactions will be attempted. Some drivers may offer an emulation of transactions, but sometimes with significant overhead, in which case the user must explicitly allow for such an emulation by first setting `$transactionsForce <- TRUE`.

Methods:

`$open(read_only)`
(Re-)opens the vector layer on the existing DSN. Use this method to open a layer that has been closed using `$close()`. May be used to re-open a layer with a different read/write access (`read_only` set to `TRUE` or `FALSE`). The method will first close an open dataset, so it is not required to call `$close()` explicitly in this case. No return value, called for side effects.

`$isOpen()`
Returns a `logical` value indicating whether the vector dataset is open.

`$getDsn()`
Returns a character string containing the `dsn` associated with this `GDALVector` object (`dsn` originally used to open the layer).

`$getFileList()`
Returns a character vector of files believed to be part of the data source. If it returns an empty string (`""`) it means there is believed to be no local file system files associated with the dataset (e.g., a virtual file system). The returned filenames will normally be relative or absolute paths depending on the path used to originally open the dataset.

`$info()`
Prints information about the vector layer to the console (no return value, called for that side effect only). For non-SQL DSN/layer, calls `ogrinfo()` passing the command options `cl_arg = c("-so", "-nomd")`, and for layers open with a SQL statement, calls `ogrinfo()` passing the command options `cl_arg = c("-so", "-nomd", "-sql", <statement>)`.

`$getDriverShortName()`
Returns the short name of the vector format driver.

`$getDriverLongName()`
Returns the long name of the vector format driver.

`$getName()`
Returns the layer name.
`$getFieldNames()`
Returns a character vector of the layer's field names.
`$testCapability()`
Tests whether the layer supports named capabilities based on the current read/write access. Returns a list of capabilities with values `TRUE` or `FALSE`. The returned list contains the following named elements: `RandomRead`, `SequentialWrite`, `RandomWrite`, `UpsertFeature`, `FastSpatialFilter`, `FastFeatureCount`, `FastGetExtent`, `FastSetNextByIndex`, `FastGetArrowStream`, `FastWriteArrowBatch`, `CreateField`, `CreateGeomField`, `DeleteField`, `ReorderFields`, `AlterFieldDefn`, `AlterGeomFieldDefn`, `DeleteFeature`, `StringsAsUTF8`, `Transactions`, `CurveGeometries`. Note that some layer capabilities are GDAL version dependent and may not be listed if not supported by the GDAL version currently in use. (See the GDAL documentation for `OGR_L_TestCapability()`.)
`$getFIDColumn()`
Returns the name of the underlying database column being used as the FID column, or empty string (`""`) if not supported.
`$getGeomType()`
Returns the well known name of the layer geometry type as character string. For layers with multiple geometry fields, this method only returns the geometry type of the first geometry column. For other columns, use `$getLayerDefn()`. For layers without any geometry field, this method returns `"NONE"`.
`$getGeometryColumn()`
Returns he name of the underlying database column being used as the geometry column, or an empty string (`""`) if not supported. For layers with multiple geometry fields, this method only returns the name of the first geometry column. For other columns, use method `$getLayerDefn()`.
`$getSpatialRef()`
Returns a WKT string containing the spatial reference system for this layer, or empty string (`""`) if no spatial reference exists.
`$bbox()`
Returns a numeric vector of length four containing the bounding box for this layer (xmin, ymin, xmax, ymax). Note that `bForce = true` is set in the underlying API call to `OGR_L_GetExtent()`, so the entire layer may be scanned to compute a minimum bounding rectangle (see `FastGetExtent` in the list returned by `$testCapability()`). Depending on the format driver, a spatial filter may or may not be taken into account, so it is safer to call `$bbox()` without setting a spatial filter.
`$getLayerDefn()`
Returns a list containing the OGR feature class definition for this layer (a.k.a. layer definition). The list contains zero or more attribute field definitions, along with one or more geometry field definitions. See ogr_define for details of the field and feature class definitions.
`$getFieldDomain(domain_name)`
Returns a list containing the definition of the OGR field domain with the passed `domain_name`, or `NULL` if `domain_name` is not found. See ogr_define for specification of the list containing a field domain definition. Some formats support the use of field domains which describe the valid values that can be stored in a given attribute field, e.g., coded values that are present in a specified enumeration, values constrained to a specified range, or values that must match a specified pattern. See `https://gdal.org/en/stable/user/vector_data_model.html#field-domains`. Requires GDAL >= 3.3.
`$setAttributeFilter(query)`
Sets an attribute query string to be used when fetching features via the `$getNextFeature()`

or $fetch() methods. Only features for which query evaluates as true will be returned. The query string should be in the format of an SQL WHERE clause, described in the "WHERE" section of the OGR SQL dialect documentation (e.g., "population > 1000000 and population < 5000000", where population is an attribute in the layer). In some cases (RDBMS backed drivers, SQLite, GeoPackage) the native capabilities of the database may be used to to interpret the WHERE clause, in which case the capabilities will be broader than those of OGR SQL. Note that installing a query string will generally result in resetting the current reading position (as with $resetReading() described below). The query parameter may be set to empty string ("") to clear the current attribute filter.

$getAttributeFilter()

Returns the attribute query string currently in use, or empty string ("") if an attribute filter is not set.

$setIgnoredFields(fields)

Set which fields can be omitted when retrieving features from the layer. The fields argument is a character vector of field names. Passing an empty string ("") for fields will reset to no ignored fields. If the format driver supports this functionality (testable using $testCapability()$IgnoreFields), it will not fetch the specified fields in subsequent calls to $getFeature() / $getNextFeature() / $fetch(), and thus save some processing time and/or bandwidth. Besides field names of the layer, the following special fields can be passed: "OGR_GEOMETRY" to ignore geometry and "OGR_STYLE" to ignore layer style. By default, no fields are ignored. Note that fields that are used in an attribute filter should generally not be set as ignored fields, as most drivers (such as those relying on the OGR SQL engine) will be unable to correctly evaluate the attribute filter. No return value, called for side effects.

$setSelectedFields(fields)

Set which fields will be included when retrieving features from the layer. The fields argument is a character vector of field names. Passing an empty string ("") for fields will reset to no ignored fields. See the $setIgnoredFields() method above for more information. The data source must provide IgnoreFields capability in order to set selected fields. Note that geometry fields, if desired, must be specified when setting selected fields, either by including named geometry field(s) or the special field "OGR_GEOMETRY" in the fields argument. No return value, called for side effects.

$getIgnoredFields()

Returns a character vector containing the list of currently ignored fields, or an empty vector (character(0)) if no fields are currently set to ignored (or if the format driver does not support ignored fields).

$setSpatialFilter(wkt)

Sets a new spatial filter from a geometry in WKT format. This method sets the geometry to be used as a spatial filter when fetching features via the $getNextFeature() or $fetch() methods. Only features that geometrically intersect the filter geometry will be returned. Currently this test may be inaccurately implemented (depending on the vector format driver), but it is guaranteed that all features whose envelope overlaps the envelope of the spatial filter will be returned. This can result in more shapes being returned that should strictly be the case. wkt is a character string containing a WKT geometry in the same coordinate system as the layer. An empty string ("") may be passed indicating that the current spatial filter should be cleared, but no new one instituted.

$setSpatialFilterRect(bbox)

Sets a new rectangular spatial filter. This method sets a rectangle to be used as a spatial filter when fetching features via the $getNextFeature() or $fetch() methods. Only features that geometrically intersect the given rectangle will be returned. bbox is a numeric vector of length four containing xmin, ymin, xmax, ymax in the same coordinate system as the layer as a whole

(as returned by `$getSpatialRef()`).

`$getSpatialFilter()`

Returns the current spatial filter geometry as a WKT string, or empty string (`""`) if a spatial filter is not set.

`$clearSpatialFilter()`

Clears a spatial filter that was set with `$setSpatialFilterRect()`. No return value, called for that side effect.

`$getFeatureCount()`

Returns the number of features in the layer. For dynamic databases the count may not be exact. This method forces a count in the underlying API call (i.e., `bForce = TRUE` in the call to `OGR_L_GetFeatureCount()`). Note that some vector drivers will actually scan the entire layer once to count features. The `FastFeatureCount` element in the list returned by the `$testCapability()` method can be checked if this might be a concern. The number of features returned takes into account the spatial and/or attribute filters. Some driver implementations of this method may alter the read cursor of the layer.

`$getNextFeature()`

Fetch the next available feature from this layer. Only features matching the current spatial and/or attribute filter (if defined) will be returned. This method implements sequential access to the features of a layer. The `$resetReading()` method can be used to start at the beginning again. Returns a list with the unique feature identifier (FID), the attribute and geometry field names, and their values. The returned list carries the `OGRFeature` class attribute with S3 methods for for `print()` and `plot()`. `NULL` is returned if no more features are available.

`$setNextByIndex(i)`

Moves the read cursor to feature `i` in the current result set (with 0-based indexing). This method allows positioning of a layer such that a call to `$getNextFeature()` or `$fetch()` will read the requested feature(s), where `i` is an absolute index into the current result set. So, setting `i = 3` would mean the next feature read with `$getNextFeature()` would have been the fourth feature read if sequential reading took place from the beginning of the layer, including accounting for spatial and attribute filters. This method is not implemented efficiently by all vector format drivers. The default implementation simply resets reading to the beginning and then calls `GetNextFeature()` `i` times. To determine if fast seeking is available on the current layer, check the `FastSetNextByIndex` element in the list returned by the `$testCapability()` method. No return value, called for side effect.

`$getFeature(fid)`

Returns a feature by its identifier. The value of `fid` must be a numeric value, optionally carrying the `bit64::integer64` class attribute. Success or failure of this operation is unaffected by any spatial or attribute filters that may be in effect. The `RandomRead` element in the list returned by `$testCapability()` can be checked to establish if this layer supports efficient random access reading; however, the call should always work if the feature exists since a fallback implementation just scans all the features in the layer looking for the desired feature. Returns a list with the unique feature identifier (FID), the attribute and geometry field names, and their values, or `NULL` on failure. Note that sequential reads (with `$getNextFeature()`) are generally considered interrupted by a call to `$getFeature()`.

`$resetReading()`

Reset feature reading to start on the first feature. No return value, called for that side effect.

`$fetch(n)`

Fetches the next `n` features from the layer and returns them as a data frame. This allows retrieving the entire set of features, one page of features at a time, or the remaining features (from the current

cursor position). Returns a data frame with as many rows as features were fetched, and as many columns as attribute plus geometry fields in the result set, even if the result is a single value or has one or zero rows. The returned data frame carries the `OGRFeatureSet` class attribute with S3 methods for for `print()` and `plot()`.

This method is an analog of `DBI::dbFetch()`.

The `n` argument is the maximum number of features to retrieve per fetch given as `integer` or `numeric` but assumed to be a whole number (will be truncated). Use `n = -1` or `n = Inf` to retrieve all pending features (resets reading to the first feature). Otherwise, `$fetch()` can be called multiple times to perform forward paging from the current cursor position. Passing `n = NA` is also supported and returns the remaining features. Fetching zero features is possible to retrieve the structure of the feature set as a data frame (columns fully typed).

OGR field types are returned as the following R types (type-specific `NA` for OGR NULL values):

- **OFTInteger**: `integer` value
- **OFTInteger subtype OFSTBoolean**: `logical` value
- **OFTIntegerList**: vector of `integer` (list column)
- **OFTInteger64**: `numeric` value carrying the `"integer64"` class attribute
- **OFTInteger64 subtype OFSTBoolean**: `logical` value
- **OFTInteger64List**: vector of `bit64::integer64` (list column)
- **OFTReal**: `numeric` value
- **OFTRealList**: vector of `numeric` (list column)
- **OFTString**: `character` string
- **OFTStringList**: vector of `character` strings (list column)
- **OFTDate**: `numeric` value of class `"Date"`
- **OFTDateTime**: `numeric` value of class `"POSIXct"` (millisecond accuracy)
- **OFTTime**: `character` string (`"HH:MM:SS"`)
- **OFTBinary**: `raw` vector (list column, `NULL` entries for OGR NULL values)

Geometries are not returned if the field `returnGeomAs` is set to `NONE`. Omitting the geometries may be beneficial for performance and memory usage when access only to feature attributes is needed. Geometries are returned as `raw` vectors in a data frame list column when `returnGeomAs` is set to `WKB` (the default) or `WKB_ISO`, or as `character` strings when `returnGeomAs` is set to one of `WKT` or `WKT_ISO`.

Note that `$getFeatureCount()` is called internally when fetching the full feature set or all remaining features (but not for a page of features).

`$getArrowStream()`

Returns a `nanoarrow_array_stream` object exposing an Arrow C stream on the layer (requires GDAL >= 3.6). The writable field `$arrowStreamOptions` can be used to set options before calling this method (see above). An error is raised if an array stream on the layer cannot be obtained. Generally, only one ArrowArrayStream can be active at a time on a given layer (i.e., the last active one must be explicitly released before a next one is asked). Changing attribute or spatial filters, ignored columns, modifying the schema or using `$resetReading()` / `$getNextFeature()` while using an ArrowArrayStream is strongly discouraged and may lead to unexpected results. As a rule of thumb, no OGRLayer methods that affect the state of a layer should be called on the layer while an ArrowArrayStream on it is active. Methods available on the stream object are: `$get_schema()`, `$get_next()` and `$release()` (see Examples).

The stream should be released once reading is complete. Calling the release method as soon as you can after consuming a stream is recommended by the **nanoarrow** documentation.

See also the $testCapability() method above to check whether the format driver provides a specialized implementation (FastGetArrowStream), as opposed to the (slower) default implementation. Note however that specialized implementations may fallback to the default when attribute or spatial filters are in use. (See the GDAL documentation for OGR_L_GetArrowStream().)

$releaseArrowStream()
Releases the Arrow C stream returned by $getArrowStream() and clears the nanoarrow_array_stream object (if GDAL >= 3.6, otherwise does nothing). This is equivalent to calling the $release() method on the nanoarrow_array_stream object. No return value, called for side effects.

$setFeature(feature)
Rewrites/replaces an existing feature. This method writes a feature based on the feature id within the input feature. The feature argument is a named list of fields and their values, and must include a FID element referencing the existing feature to rewrite. The RandomWrite element in the list returned by $testCapability() can be checked to establish if this layer supports random access writing via $setFeature(). The way omitted fields in the passed feature are processed is driver dependent:

- SQL-based drivers which implement set feature through SQL UPDATE will skip unset fields, and thus the content of the existing feature will be preserved.
- The shapefile driver will write a NULL value in the DBF file.
- The GeoJSON driver will take into account unset fields to remove the corresponding JSON member.

Returns logical TRUE upon successful completion, or FALSE if setting the feature did not succeed. The FID of the last feature written to the layer may be obtained with the method $getLastWriteFID() (see below). To set a feature, but create it if it doesn't exist see the $upsertFeature() method.

$createFeature(feature)
Creates and writes a new feature within the layer. The feature argument is a named list of fields and their values (might be one row of a data frame). The passed feature is written to the layer as a new feature, rather than overwriting an existing one. If the feature has a FID element with other than NA (i.e., a numeric value, optionally carrying the bit64::integer64 class attribute and assumed to be a whole number), then the format driver may use that as the feature id of the new feature, but not necessarily. The FID of the last feature written to the layer may be obtained with the method $getLastWriteFID() (see below). Returns logical TRUE upon successful completion, or FALSE if creating the feature did not succeed. To create a feature, but set it if it already exists see the $upsertFeature() method.

$batchCreateFeature(feature_set)
Batch version of $createFeature(). Creates and writes a batch of new features within the layer from input passed as a data frame in the feature_set argument. Column names in the data frame must match field names of the layer and have compatible data types. The specifications listed above under the $fetch() method generally apply to input data types for writing, but integers may be passed as 'numeric', and the 'integer64' class attribute is not strictly required on 'numeric' input if it is not needed for the data being passed to an OFTInteger64 field. Returns a logical vector of length equal to the number of input features (rows of the data frame), with TRUE indicating success for the feature at that row index, or FALSE if writing the feature failed. It is recommended to use transactions when batch writing features to a layer (see $startTransaction() below). This will generally give large performance benefit with data sources that provide efficient transaction support (e.g., RDBMS-based sources such as GeoPackage and PostGIS). In addition, the return value of $batchCreateFeature() can be checked, and the transaction optionally committed or rolled back based on results of the operation across the full set of input features.

`$upsertFeature(feature)`
Rewrites/replaces an existing feature or creates a new feature within the layer. This method will write a feature to the layer, based on the feature id within the input feature. The `feature` argument is a named list of fields and their values (might be one row of a data frame), potentially including a `FID` element referencing an existing feature to rewrite. If the feature id doesn't exist a new feature will be written. Otherwise, the existing feature will be rewritten. The `UpsertFeature` element in the list returned by `$testCapability()` can be checked to determine if this layer supports upsert writing. See `$setFeature()` above for a description of how omitted fields in the passed feature are processed. Returns logical `TRUE` upon successful completion, or `FALSE` if upsert did not succeed. Requires GDAL >= 3.6.

`$getLastWriteFID()`
Returns the FID of the last feature written (either newly created or updated existing). `NULL` is returned if no features have been written in the layer. Note that OGRNullFID (`-1`) may be returned after writing a feature in some formats. This is the case if a FID has not been assigned yet, and generally does not indicate an error (e.g., formats that do not store a persistent FID and assign FIDs upon a sequential read operation). The returned FID is a numeric value carrying the `bit64::integer64` class attribute.

`$deleteFeature(fid)`
Deletes a feature from the layer. The feature with the indicated feature ID is deleted from the layer if supported by the format driver. The value of `fid` must be a numeric value, optionally carrying the `bit64::integer64` class attribute (should be a whole number, will be truncated). The `DeleteFeature` element in the list returned by `$testCapability()` can be checked to establish if this layer has delete feature capability. Returns logical `TRUE` if the operation succeeds, or `FALSE` on failure.

`$syncToDisk()`
Flushes pending changes to disk. This call is intended to force the layer to flush any pending writes to disk, and leave the disk file in a consistent state. It would not normally have any effect on read-only datasources. Some formats do not implement this method, and will still return no error. An error is only returned if an error occurs while attempting to flush to disk. In any event, you should always close any opened datasource with `$close()` which will ensure all data is correctly flushed. Returns logical `TRUE` if no error occurs (even if nothing is done) or `FALSE` on error.

`$startTransaction()`
Creates a transaction if supported by the vector data source. By default, only "efficient" transactions will be attempted. See the writable field `$transactionsForce` above, which must be set to `TRUE` to allow for emulated transactions. These are supported by some drivers but with potentially significant overhead. The function `ogr_ds_test_cap()` can be used to determine whether a vector data source supports efficient or emulated transactions.

All changes done after the start of the transaction are definitely applied in the data source if `$commitTransaction()` is called. They can be canceled by calling `$rollbackTransaction()` instead. Nested transactions are not supported. Transactions are implemented at the dataset level, so multiple `GDALVector` objects using the same data source should not have transactions active at the same time.

In case `$startTransaction()` fails, neither `$commitTransaction()` nor `$rollbackTransaction()` should be called. If an error occurs after a successful `$startTransaction()`, the whole transaction may or may not be implicitly canceled, depending on the format driver (e.g., the PostGIS driver will cancel it, SQLite/GPKG will not). In any case, in the event of an error, an explicit call to `$rollbackTransaction()` should be done to keep things balanced.

Returns logical TRUE if the transaction is created, or FALSE on failure.

`$commitTransaction()`
Commits a transaction if supported by the vector data source. Returns a logical value, TRUE if the transaction is successfully committed. Returns FALSE if no transaction is active, or the rollback fails, or if the data source does not support transactions. Depending on the format driver, this may or may not abort layer sequential reading that may be active.

`$rollbackTransaction()`
Rolls back a data source to its state before the start of the current transaction, if transactions are supported by the data source. Returns a logical value, TRUE if the transaction is successfully rolled back. Returns FALSE if no transaction is active, or the rollback fails, or if the data source does not support transactions.

`$getMetadata()`
Returns a character vector of all metadata NAME=VALUE pairs for the layer or empty string ("") if there are no metadata items.

`$setMetadata(metadata)`
Sets metadata on the layer if the format supports it. The `metadata` argument is given as a character vector of NAME=VALUE pairs. Returns logical TRUE on success or FALSE if metadata could not be set.

`$getMetadataItem(mdi_name)`
Returns the value of a specific metadata item named `mdi_name`, or empty string ("") if no matching item is found.

`$close()`
Closes the vector dataset (no return value, called for side effects). Calling `$close()` results in proper cleanup, and flushing of any pending writes. The `GDALVector` object is still available after calling `$close()`. The layer can be re-opened on the existing `dsn` with `$open(read_only = TRUE|FALSE)`.

See Also

ogr_define, ogr_manage, `ogr2ogr()`, `ogrinfo()`

GDAL vector format descriptions:
https://gdal.org/en/stable/drivers/vector/index.html

GDAL-supported SQL dialects:
https://gdal.org/en/stable/user/ogr_sql_sqlite_dialect.html

GDAL Vector API documentation:
https://gdal.org/en/stable/api/index.html

Examples

```
## MTBS fire perimeters in Yellowstone National Park 1984-2022
f <- system.file("extdata/ynp_fires_1984_2022.gpkg", package = "gdalraster")

## copy to a temporary file that is writeable
dsn <- file.path(tempdir(), basename(f))
file.copy(f, dsn)

(lyr <- new(GDALVector, dsn, "mtbs_perims"))
```

```
str(lyr)

## dataset info
lyr$getDriverShortName()
lyr$getDriverLongName()
lyr$getFileList()

## layer info
lyr$getName()
lyr$getGeomType()
lyr$getGeometryColumn()
lyr$getFIDColumn()
lyr$getSpatialRef()
lyr$bbox()

## layer capabilities
lyr$testCapability()

## re-open with write access
lyr$open(read_only = FALSE)
lyr$testCapability()$SequentialWrite
lyr$testCapability()$RandomWrite

## feature class definition - a list of field names and their definitions
defn <- lyr$getLayerDefn()
names(defn)
str(defn)

## default value of the read/write field 'returnGeomAs'
lyr$returnGeomAs

lyr$getFeatureCount()

## sequential read cursor
# a single feature returned as a named list of fields and their values:
(feat <- lyr$getNextFeature())

## set an attribute filter
lyr$setAttributeFilter("ig_year = 2020")
lyr$getFeatureCount()

feat <- lyr$getNextFeature()
plot(feat)

## NULL when no more features are available
lyr$getNextFeature()

## reset reading to the start
lyr$resetReading()
lyr$getNextFeature()

## clear the attribute filter
lyr$setAttributeFilter("")
```

```
lyr$getFeatureCount()

## set a spatial filter
## get the bounding box of the largest 1988 fire and use as spatial filter
## first set a temporary attribute filter to do the lookup
lyr$setAttributeFilter("ig_year = 1988 ORDER BY burn_bnd_ac DESC")
(feat <- lyr$getNextFeature())

bbox <- g_wk2wk(feat$geom) |> bbox_from_wkt()

## set spatial filter on the full layer
lyr$setAttributeFilter("")  # clears the attribute filter
lyr$setSpatialFilterRect(bbox)
lyr$getFeatureCount()

## fetch in chunks and return as data frame (class OGRFeatureSet)
feat_set <- lyr$fetch(20)
head(feat_set)
plot(feat_set)

## the next chunk
feat_set <- lyr$fetch(20)
nrow(feat_set)

## no features remaining
feat_set <- lyr$fetch(20)
nrow(feat_set)
str(feat_set)  # 0-row data frame with columns fully typed

## or, fetch all pending features from the beginning
feat_set <- lyr$fetch(-1)  # resets reading to the first feature
nrow(feat_set)
plot(feat_set)

lyr$clearSpatialFilter()
lyr$getFeatureCount()

lyr$close()

## simple example of feature write methods showing use of various data types
## create and write to a new layer in a GeoPackage data source
dsn2 <- tempfile(fileext = ".gpkg")

## define a feature class
defn <- ogr_def_layer("POINT", srs = epsg_to_wkt(4326))

## add field definitions
defn$unique_int <- ogr_def_field("OFTInteger", is_nullable = FALSE,
                                 is_unique = TRUE)
defn$bool_data <- ogr_def_field("OFTInteger", fld_subtype = "OFSTBoolean")
defn$large_ints <- ogr_def_field("OFTInteger64")
defn$doubles <- ogr_def_field("OFTReal")
defn$strings <- ogr_def_field("OFTString", fld_width = 50)
```

```
defn$dates <- ogr_def_field("OFTDate")
defn$dt_modified <- ogr_def_field("OFTDateTime",
                                  default_value = "CURRENT_TIMESTAMP")
defn$blobs <- ogr_def_field("OFTBinary")

lyr <- ogr_ds_create("GPKG", dsn2, "test_layer", layer_defn = defn,
                     return_obj = TRUE)

# lyr$getLayerDefn() |> str()

## define a feature to write
feat1 <- list()
# $FID is omitted since it is assigned when written (could also be NA)
# $dt_modified is omitted since a default timestamp is defined on the field
feat1$unique_int <- 1001
feat1$bool_data <- TRUE
# pass a string to as.integer64() since the value is too large to be
# represented exactly as an R numeric value (i.e., double)
feat1$large_ints <- bit64::as.integer64("90071992547409910")
feat1$doubles <- 1.234
feat1$strings <- "A test string"
feat1$dates <- as.Date("2025-01-01")
feat1$blobs <- charToRaw("A binary object")
feat1$geom <- "POINT (1 1)"  # can be a WKT string or raw vector of WKB

## create as a new feature in the layer
lyr$createFeature(feat1)

## get the assigned FID
lyr$getLastWriteFID()

## attempt to re-write the same feature fails due to the unique constraint
lyr$createFeature(feat1)

feat2 <- list()
feat2$unique_int <- 1002
feat2$bool_data <- FALSE
feat2$large_ints <- bit64::as.integer64("90071992547409920")
feat2$doubles <- 2.345
feat2$strings <- "A test string 2"
feat2$dates <- as.Date("2024-01-02")
feat2$blobs <- charToRaw("A binary object 2")
feat2$geom <- "POINT (2 2)"

lyr$createFeature(feat2)
lyr$getLastWriteFID()

## close and re-open as a read-only layer
lyr$open(read_only = TRUE)

lyr$getFeatureCount()
feat_set <- lyr$fetch(-1)  # -1 to fetch all features from the beginning
str(feat_set)
```

```
## edit an existing feature, e.g., feat <- lyr$getFeature(2)
## here we copy a row of the data frame returned by lyr$fetch() above
feat <- feat_set[2,]
str(feat)

Sys.sleep(1)  # to ensure a timestamp difference

feat$bool_data <- TRUE
feat$strings <- paste(feat$strings, "- edited")
feat$dt_modified <- Sys.time()
feat$geom <- "POINT (2.001 2.001)"

lyr$open(read_only = FALSE)

## lyr$setFeature() re-writes the feature identified by the $FID element
lyr$setFeature(feat)

lyr$open(read_only = TRUE)
lyr$getFeatureCount()

lyr$returnGeomAs <- "WKT"
feat_set <- lyr$fetch(-1)
str(feat_set)

lyr$close()

## Arrow array stream exposed as a nanoarrow_array_stream object
## requires GDAL >= 3.6
if (gdal_version_num() >= gdal_compute_version(3, 6, 0)) {

  sql <- "SELECT incid_name, geom from mtbs_perims LIMIT 5"
  lyr <- new(GDALVector, dsn, sql)

  stream <- lyr$getArrowStream()

  # disable a warning for the example that can be safely ignored here
  options(nanoarrow.warn_unregistered_extension = FALSE)

  # pull all the batches into a data frame
  d <- as.data.frame(stream)

  # the geometry column is a list column of WKB raw vectors, which can be
  # passed to the Geometry API g_*() functions, e.g.,
  g_centroid(d$geom) |> print()

  # release the stream when finished
  stream$release()

  lyr$close()
}
```

gdal_cli *Functions for using GDAL CLI algorithms*

Description

This set of functions can be used to access and run gdal command line interface (CLI) algorithms.

Requires GDAL >= 3.11.3

Experimental (see the section Development Status below)

Usage

```
gdal_commands(contains = NULL, recurse = TRUE, cout = TRUE)

gdal_usage(cmd = NULL)

gdal_run(cmd, args, setVectorArgsFromObject = TRUE)

gdal_alg(cmd = NULL, args = NULL, parse = TRUE)

gdal_global_reg_names()
```

Arguments

contains	Optional character string for filtering output to certain commands, e.g., gdal_commands("vector").
recurse	Logical value, TRUE to include all subcommands recursively (the default). Set to FALSE to include only the top-level gdal commands (i.e., raster, vector, etc.)
cout	Logical value, TRUE to print a list of commands along with their descriptions and help URLS to the console (the default).
cmd	A character string or character vector containing the path to the algorithm, e.g., "raster reproject" or c("raster", "reproject"). Defaults to "gdal", the main entry point to CLI commands.
args	Either a character vector or a named list containing input arguments of the algorithm (see section Algorithm Argument Syntax below).
setVectorArgsFromObject	Logical value, TRUE to set algorithm arguments automatically when the "input" argument or the "like" argument is an object of class GDALVector (the default). Can be set to FALSE to disable automatically setting algorithm arguments from GDALVector input (see Note).
parse	Logical value, TRUE to attempt parsing args if they are given in gdal_alg() (the default). Set to FALSE to instantiate the algorithm without parsing arguments. The $parseCommandLineArgs() method on the returned object can be called to parse arguments and obtain the result of that, with potentially useful error messages.

Details

These functions provide an interface to GDAL CLI algorithms by way of the C++ exposed class GDALAlg. See the class documentation for additional information (?GDALAlg).

gdal_commands() prints a list of commands and their descriptions to the console, and returns (invisibly) a data frame with columns command, description, URL. The contains argument can be used to filter the output, e.g., gdal_commands("vector") to return only commands for working with vector inputs.

gdal_usage() prints a help message to the console for a given command, or for the root gdal entry point if called with no argument. No return value, called for that side effect only.

gdal_run() executes a GDAL CLI algorithm and returns it as an object of class GDALAlg. A list containing algorithm output(s) can be accessed by calling the $outputs() method (plural) on the returned object, or, more conveniently in most cases, by calling $output() (singular) to return the the single output value when there is only one. After assigning the output, or otherwise completing work with the GDALAlg object, its $release() method can be called to close datasets and free resources.

gdal_alg() instantiates and returns an object of class GDALAlg without running it. Passing argument values to the requested CLI algorithm is optional. This function may be useful (with or without argument values) for obtaining algorithm properties with the returned object's $info() method, obtaining properties of algorithm arguments ($argInfo(arg_name)), or obtaining algorithm usage as a JSON-formatted string ($usageAsJSON()). This function is simply an alternative to calling the new() constructor for class GDALAlg. Executing the returned algorithm is optional by calling the object's $run() method (assuming argument values were given).

gdal_global_reg_names() returns a character vector containing the names of the algorithms in the GDAL global algorithm registry. These are the top-level nodes (raster, vector, etc.) known to GDAL. Potentially code external to GDAL could register a new command available for CLI use in a GDAL plugin. This function may be useful in certain troubleshooting scenarios. It will return a vector of length zero if no names are returned from the global registry.

Algorithm Argument Syntax

Arguments are given in R as a character vector or named list, but otherwise syntax basically matches the GDAL specification for arguments as they are given on the command line. Those specifications are listed here along with some amendments regarding the character vector and named list formats. Programmatic usage also allows passing and receiving datasets as objects (i.e., GDALRaster or GDALVector), in addition to dataset names (e.g., filename, URL, database connection string).

- Commands accept one or several positional arguments, typically for dataset names (or in R as GDALRaster or GDALVector datasets). The order is input(s) first, output last. Positional arguments can also be specified as named arguments, if preferred to avoid any ambiguity.
- Named arguments have:
  - at least one "long" name, preceded by two dash characters
  - optionally, auxiliary long names (i.e., aliases),
  - and optionally a one-letter short name, preceded by a single dash character, e.g., -f, --of, --format, --output-format <OUTPUT-FORMAT>
- Boolean arguments are specified by just specifying the argument name in character vector format. In R list format, the named element must be assigned a value of logical TRUE.

- Arguments that require a value are specified like:
  - `-f VALUE` for one-letter short names
  - `--format VALUE` or `--format=VALUE` for long names
  - in a named list, this might look like: `args$format <- VALUE`
- Some arguments can be multi-valued. Some of them require all values to be packed together and separated with comma. This is, e.g., the case of:
  `--bbox <BBOX> Clipping bounding box as xmin,ymin,xmax,ymax`
  e.g., `--bbox=2.1,49.1,2.9,49.9`
- Others accept each value to be preceded by a new mention of the argument name, e.g., `c("--co", "COMPRESS=LZW", "--co", "TILED=YES")`. For that one, if the value of the argument does not contain commas, the packed form is also accepted: `--co COMPRESS=LZW,TILED=YES`. Note that repeated mentions of an argument are possible in the character vector format for argument input, whereas arguments given in named list format must use argument long names as the list element names, and the packed format for the values (which can be a character vector or numeric vector of values).
- Named arguments can be placed before or after positional arguments.

Development Status

The GDAL Command Line Interface Modernization was first introduced in the GDAL 3.11.0 release (2025-05-09). The GDAL project provides warning that the new CLI "is provisionally provided as an alternative interface to GDAL and OGR command line utilities. The project reserves the right to modify, rename, reorganize, and change the behavior until it is officially frozen via PSC vote in a future major GDAL release. The utility needs time to mature, benefit from incremental feedback, and explore enhancements without carrying the burden of full backward compatibility. Your usage of it should have no expectation of compatibility until that time." (https://gdal.org/en/latest/programs/#gdal-application)

Initial bindings to enable programmatic use of the CLI algorithms from R were added in **gdalraster** 2.2.0, and will evolve over future releases. *The bindings are considered experimental until the upstream API is declared stable*. Breaking changes in minor version releases are possible until then. Please use with those cautions in mind. Bug reports may be filed at: https://github.com/USDAForestService/gdalraster/issues.

Note

Commands do not require the leading `"gdal"` root node. They may begin with a top-level command (e.g., `"raster"`, `"vector"`, etc.).

When using argument names as the element names of a list, the underscore character can be substituted for the dash characters that are used in some names. This avoids having to surround names in backticks when they are used to access list elements in the form `args$arg_name` (the form `args[["arg-name"]]` also works).

When `setVectorArgsFromObject` is `TRUE` (the default) and the `"input"` or `"like"` argument for an algorithm is given as a `GDALVector` object, corresponding algorithm arguments will be set automatically based on properties of the object (when the argument is available to the algorithm):

- `"input-format"`: set to the `GDALVector` object's driver short name
- `"input-layer"`: set to the `GDALVector` layer name if it is not a SQL layer

- "sql": set to the SQL statement if the GDALVector layer is defined by one
- "dialect": set to the SQL dialect if one is specified for a SQL layer
- "like-layer": set to the GDALVector layer name if it is not a SQL layer
- "like-sql": set to the SQL statement if the GDALVector layer is defined by one

Argument values specified explicitly will override the automatic setting (as long as they result in a parsable set of arguments). If setVectorArgsFromObject is FALSE, then only the vector dataset is passed to the algorithm, i.e., without automatically passing any layer specifications.

See Also

GDALAlg-class

gdal Command Line Interface (CLI)
https://gdal.org/en/stable/programs/index.html

Using gdal CLI algorithms from R
https://usdaforestservice.github.io/gdalraster/articles/use-gdal-cli-from-r.html

Examples

```
## top-level commands
gdal_commands(recurse = FALSE)

## convert storml_elev.tif to GeoPackage raster
gdal_commands("convert")

gdal_usage("raster convert")

f_tif <- system.file("extdata/storml_elev.tif", package="gdalraster")
f_gpkg <- file.path(tempdir(), "storml_elev.gpkg")

args <- c("--overwrite", f_tif, f_gpkg)
(alg <- gdal_run("raster convert", args))

(ds <- alg$output())

alg$release()

plot_raster(ds, legend = TRUE)

ds$close()
unlink(f_gpkg)

## get help for vector commands
gdal_usage("vector")

## clip a vector layer by a bounding box
gdal_usage("vector clip")

f <- system.file("extdata/ynp_fires_1984_2022.gpkg", package="gdalraster")
f_clip <- file.path(tempdir(), "ynp_fires_clip.gpkg")
```

```
# some multi-valued arguments require all values packed and comma separated
# e.g., --bbox <BBOX>
bb <- c(469686, 11442, 544070, 85508)
bb <- paste(bb, collapse = ",")

args <- c("--bbox", bb, "--overwrite", f, f_clip)
(alg <- gdal_run("vector clip", args))

(lyr <- alg$output())

lyr$bbox()

lyr$getFeatureCount()

lyr$close()
alg$release()
unlink(f_clip)

## rasterize a vector layer and return output as a GDALRaster object
gdal_usage("vector rasterize")

f_out = file.path(tempdir(), "ynp_fire_year.tif")

# arguments in list format
args <- list(input = f,
             sql = "SELECT * FROM mtbs_perims ORDER BY ig_year",
             attribute_name = "ig_year",
             output = f_out,
             overwrite = TRUE,
             creation_option = c("TILED=YES", "COMPRESS=DEFLATE"),
             resolution = c(90, 90),
             output_data_type = "Int16",
             init = -32767,
             nodata = -32767)

(alg <- gdal_run("vector rasterize", args))

(ds <- alg$output())

alg$release()

pal <- scales::viridis_pal(end = 0.8, direction = -1)(6)
ramp <- scales::colour_ramp(pal)
plot_raster(ds, legend = TRUE, col_map_fn = ramp, na_col = "#d9d9d9",
            main = "YNP Fires 1984-2022 - Most Recent Burn Year")

ds$close()
deleteDataset(f_out)

## pipeline syntax
# "raster pipeline" example 2 from:
# https://gdal.org/en/latest/programs/gdal_raster_pipeline.html
```

```
# serialize the command to reproject a GTiff file into GDALG format, and
# then later read the GDALG file
# GDAL Streamed Algorithm format:
# https://gdal.org/en/stable/drivers/raster/gdalg.html

gdal_usage("raster pipeline")

f_tif <- system.file("extdata/storml_elev.tif", package="gdalraster")
f_out <- file.path(tempdir(), "storml_elev_epsg_32100.gdalg.json")

args <- c("read", "--input", f_tif, "!",
          "reproject", "--dst-crs=EPSG:32100", "!",
          "write", "--output", f_out, "--overwrite")

alg <- gdal_run("raster pipeline", args)
alg$release()

# content of the .gdalg.json file
readLines(f_out, warn = FALSE) |> writeLines()

(ds <- new(GDALRaster, f_out))

plot_raster(ds, legend = TRUE)

ds$close()
unlink(f_out)
```

---

gdal_compute_version *Compute a GDAL integer version number from major, minor, revision*

Description

gdal_compute_version() computes a full integer version number (GDAL_VERSION_NUM) from individual components (major, minor, revision). Convenience function for checking a GDAL version requirement using gdal_version_num().

Usage

```
gdal_compute_version(maj, min, rev)
```

Arguments

maj	Numeric value, major version component (coerced to integer by truncation).
min	Numeric value, min version component (coerced to integer by truncation).
rev	Numeric value, revision version component (coerced to integer by truncation).

Value

Integer version number compatible with gdal_version_num().

See Also

gdal_version_num()

Examples

```
(gdal_version_num() >= gdal_compute_version(3, 7, 0))
```

---

gdal_formats *Retrieve information on GDAL format drivers for raster and vector*

Description

gdal_formats() returns a table of the supported raster and vector formats, with information about the capabilities of each format driver.

Usage

```
gdal_formats(format = "")
```

Arguments

format	A character string containing a driver short name. By default, information for all configured raster and vector format drivers will be returned.

Value

A data frame containing the format short name, long name, raster (logical), vector (logical), read/write flag (ro is read-only, w supports CreateCopy, w+ supports Create), virtual I/O supported (logical), and subdatasets (logical).

Note

Virtual I/O refers to operations on GDAL Virtual File Systems. See https://gdal.org/en/stable/user/virtual_file_systems.html#virtual-file-systems.

Examples

```
nrow(gdal_formats())
head(gdal_formats())

gdal_formats("GPKG")
```

gdal_get_driver_md *Get metadata for a GDAL format driver*

Description

gdal_get_driver_md() returns metadata for a driver.

Usage

```
gdal_get_driver_md(format, mdi_name = "")
```

Arguments

format	Character string giving a format driver short name (e.g., "GTiff").
mdi_name	Optional character string giving the name of a specific metadata item. Defaults to empty string ("") meaning fetch all metadata items.

Value

Either a named list of metadata items and their values as character strings, or a single character string if mdi_name is specified. Returns NULL if no metadata items are found for the given inputs.

See Also

getCreationOptions()

Examples

```
dmd <- gdal_get_driver_md("GTiff")
str(dmd)
```

gdal_version *Get GDAL version*

Description

gdal_version() returns a character vector of GDAL runtime version information. gdal_version_num() returns only the full version number (gdal_version()[2]) as an integer value.

Usage

```
gdal_version()

gdal_version_num()
```

Value

`gdal_version()` returns a character vector of length four containing:

- "-version" - one line version message, e.g., "GDAL 3.6.3, released 2023/03/12"
- "GDAL_VERSION_NUM" - formatted as a string, e.g., "3060300" for GDAL 3.6.3.0
- "GDAL_RELEASE_DATE" - formatted as a string, e.g., "20230312"
- "GDAL_RELEASE_NAME" - e.g., "3.6.3"

`gdal_version_num()` returns `as.integer(gdal_version()[2])`

Examples

```
gdal_version()

gdal_version_num()
```

---

geos_version *Get GEOS version*

Description

`geos_version()` returns version information for the GEOS library in use by GDAL. Requires GDAL >= 3.4.

Usage

```
geos_version()
```

Value

A list of length four containing:

- `name` - a string formatted as "major.minor.patch"
- `major` - major version as integer
- `minor` - minor version as integer
- `patch` - patch version as integer

List elements will be `NA` if GDAL < 3.4.

See Also

`gdal_version()`, `proj_version()`

Examples

```
geos_version()
```

`getCreationOptions` *Return the list of creation options for a GDAL driver*

Description

`getCreationOptions()` returns the list of creation options supported by a GDAL format driver. This function is a wrapper of `GDALGetDriverCreationOptionList()` in the GDAL API, parsing its XML output into a named list.

Usage

```
getCreationOptions(format, filter = NULL)
```

Arguments

`format`	Format short name (e.g., `"GTiff"`).
`filter`	Optional character vector of creation option names.

Details

The output is a nested list with names matching the creation option names. The information for each creation option is a named list with the following elements:

- `type`: a character string describing the data type, e.g., `"int"`, `"float"`, `"string"`. The type `"string-select"` denotes a list of allowed string values which are returned as a character vector in the `values` element (see below).
- `description`: a character string describing the option, or `NA` if no description is provided by the GDAL driver.
- `default`: the default value of the option as either a character string or numeric value, or `NA` if no description is provided by the GDAL driver.
- `values`: a character vector of allowed string values for the creation option if `type` is `"string-select"`, otherwise `NULL` if the option is not a `"string-select"` type.
- `min`: (GDAL >= 3.11) the minimum value of the valid range for the option, or `NA` if not provided by the GDAL driver or the option is not a numeric type.
- `max`: (GDAL >= 3.11) the maximum value of the valid range for the option, or `NA` if not provided by the GDAL driver or the option is not a numeric type.

Value

A named list with names matching the creation option names, and each element a named list with elements `type`, `description`, `default` and `values` (see Details).

See Also

`create()`, `createCopy()`, `translate()`, `validateCreationOptions()`, `warp()`

Examples

```
opt <- getCreationOptions("GTiff", "COMPRESS")
names(opt)

(opt$COMPRESS$type == "string-select")  # TRUE
opt$COMPRESS$values

all_opt <- getCreationOptions("GTiff")
names(all_opt)

# $description and $default will be NA if no value is provided by the driver
# $values will be NULL if the option is not a 'string-select' type

all_opt$PREDICTOR

all_opt$BIGTIFF
```

---

get_cache_max *Get the maximum memory size available for the GDAL block cache*

Description

`get_cache_max()` returns the maximum amount of memory available to the GDALRasterBlock caching system for caching raster read/write data. Wrapper of `GDALGetCacheMax64()` with return value in MB by default.

Usage

```
get_cache_max(units = "MB")
```

Arguments

`units`	Character string specifying units for the return value. One of `"MB"` (the default), `"GB"`, `"KB"` or `"bytes"` (values of `"byte"`, `"B"` and empty string `""` are also recognized to mean bytes).

Details

The first time this function is called, it will read the `GDAL_CACHEMAX` configuration option to initialize the maximum cache memory. The value of the configuration option can be expressed as x% of the usable physical RAM (which may potentially be used by other processes). Otherwise it is expected to be a value in MB. As of GDAL 3.10, the default value, if `GDAL_CACHEMAX` has not been set explicitly, is 5% of usable physical RAM.

Value

A numeric value carrying the `integer64` class attribute. Maximum cache memory available in the requested units.

Note

The value of the GDAL_CACHEMAX configuration option is only consulted the first time the cache size is requested (i.e., it must be set as a configuration option prior to any raster I/O during the current session). To change this value programmatically during operation of the program it is better to use set_cache_max() (in which case, always given in bytes).

See Also

GDAL_CACHEMAX configuration option

get_config_option(), set_config_option(), get_usable_physical_ram(), get_cache_used(), set_cache_max()

Examples

```
get_cache_max()
```

---

get_cache_used *Get the size of memory in use by the GDAL block cache*

Description

get_cache_used() returns the amount of memory currently in use for GDAL block caching. Wrapper of GDALGetCacheUsed64() with return value in MB by default.

Usage

```
get_cache_used(units = "MB")
```

Arguments

units	Character string specifying units for the return value. One of "MB" (the default), "GB", "KB" or "bytes" (values of "byte", "B" and empty string "" are also recognized to mean bytes).

Value

A numeric value carrying the integer64 class attribute. Amount of the available cache memory currently in use in the requested units.

See Also

GDAL Block Cache

get_cache_max(), set_cache_max()

Examples

```
get_cache_used()
```

get_config_option *Get GDAL configuration option*

Description

get_config_option() gets the value of GDAL runtime configuration option. Configuration options are essentially global variables the user can set. They are used to alter the default behavior of certain raster format drivers, and in some cases the GDAL core. For a full description and listing of available options see https://gdal.org/en/stable/user/configoptions.html.

Usage

```
get_config_option(key)
```

Arguments

key	Character name of a configuration option.

Value

Character. The value of a (key, value) option previously set with set_config_option(). An empty string ("") is returned if key is not found.

See Also

set_config_option()

vignette("gdal-config-quick-ref")

Examples

```
## this option is set during initialization of the gdalraster package
get_config_option("OGR_CT_FORCE_TRADITIONAL_GIS_ORDER")
```

get_num_cpus *Get the number of processors detected by GDAL*

Description

get_num_cpus() returns the number of processors detected by GDAL. Wrapper of CPLGetNumCPUs() in the GDAL Common Portability Library.

Usage

```
get_num_cpus()
```

Value

Integer scalar, number of CPUs.

Examples

```
get_num_cpus()
```

`get_pixel_line` *Raster pixel/line from geospatial x,y coordinates*

Description

`get_pixel_line()` converts geospatial coordinates to pixel/line (raster column, row numbers). The upper left corner pixel is the raster origin (0,0) with column, row increasing left to right, top to bottom.

Usage

```
get_pixel_line(xy, gt)
```

Arguments

`xy`	Numeric matrix of geospatial x, y coordinates in the same spatial reference system as `gt` (or two-column data frame that will be coerced to numeric matrix, or a vector x, y for one coordinate).
`gt`	Either a numeric vector of length six containing the affine geotransform for the raster, or an object of class `GDALRaster` from which the geotransform will be obtained (see Note).

Value

Integer matrix of raster pixel/line.

Note

This function applies the inverse geotransform to the input points. If `gt` is given as the numeric vector, no bounds checking is done (i.e., min pixel/line could be less than zero and max pixel/line could be greater than the raster x/y size). If `gt` is obtained from an object of class `GDALRaster`, then `NA` is returned for points that fall outside the raster extent and a warning emitted giving the number points that were outside. This latter case is equivalent to calling the `$get_pixel_line()` class method on the `GDALRaster` object (see Examples).

See Also

`GDALRaster$getGeoTransform()`, `inv_geotransform()`

Examples

```
pt_file <- system.file("extdata/storml_pts.csv", package="gdalraster")
# id, x, y in NAD83 / UTM zone 12N
pts <- read.csv(pt_file)
print(pts)

raster_file <- system.file("extdata/storm_lake.lcp", package="gdalraster")
ds <- new(GDALRaster, raster_file)
gt <- ds$getGeoTransform()
get_pixel_line(pts[, -1], gt)

# or, using the class method
ds$get_pixel_line(pts[, -1])

# add a point outside the raster extent
pts[11, ] <- c(11, 323318, 5105104)
get_pixel_line(pts[, -1], gt)

# with bounds checking on the raster extent
ds$get_pixel_line(pts[, -1])

ds$close()
```

`get_usable_physical_ram`

Get usable physical RAM reported by GDAL

Description

`get_usable_physical_ram()` returns the total physical RAM, usable by a process, in bytes. It will limit to 2 GB for 32 bit processes. Starting with GDAL 2.4.0, it will also take into account resource limits (virtual memory) on Posix systems. Starting with GDAL 3.6.1, it will also take into account RLIMIT_RSS on Linux. Wrapper of `CPLGetUsablePhysicalRAM()` in the GDAL Common Portability Library.

Usage

```
get_usable_physical_ram()
```

Value

Numeric scalar, number of bytes as `bit64::integer64` type (or 0 in case of failure).

Note

This memory may already be partly used by other processes.

Examples

```
get_usable_physical_ram()
```

g_binary_op *Binary operations on WKB or WKT geometries*

Description

These functions implement operations on pairs of geometries in OGC WKB or WKT format.

Usage

```
g_intersection(
  this_geom,
  other_geom,
  as_wkb = TRUE,
  as_iso = FALSE,
  byte_order = "LSB",
  quiet = FALSE
)

g_union(
  this_geom,
  other_geom,
  as_wkb = TRUE,
  as_iso = FALSE,
  byte_order = "LSB",
  quiet = FALSE
)

g_difference(
  this_geom,
  other_geom,
  as_wkb = TRUE,
  as_iso = FALSE,
  byte_order = "LSB",
  quiet = FALSE
)

g_sym_difference(
  this_geom,
  other_geom,
  as_wkb = TRUE,
  as_iso = FALSE,
  byte_order = "LSB",
  quiet = FALSE
)
```

Arguments

this_geom	Either a raw vector of WKB or list of raw vectors, or a character vector containing one or more WKT strings.
other_geom	Either a raw vector of WKB or list of raw vectors, or a character vector containing one or more WKT strings. Must contain the same number of geometries as this_geom.
as_wkb	Logical value, TRUE to return the output geometry in WKB format (the default), or FALSE to return as WKT.
as_iso	Logical value, TRUE to export as ISO WKB/WKT (ISO 13249 SQL/MM Part 3), or FALSE (the default) to export as "Extended WKB/WKT".
byte_order	Character string specifying the byte order when output is WKB. One of "LSB" (the default) or "MSB" (uncommon).
quiet	Logical value, TRUE to suppress warnings. Defaults to FALSE.

Details

These functions use the GEOS library via GDAL headers.

g_intersection() returns a new geometry which is the region of intersection of the two geometries operated on. g_intersects() can be used to test if two geometries intersect.

g_union() returns a new geometry which is the region of union of the two geometries operated on.

g_difference() returns a new geometry which is the region of this geometry with the region of the other geometry removed.

g_sym_difference() returns a new geometry which is the symmetric difference of this geometry and the other geometry (union minus intersection).

Value

A geometry as WKB raw vector or WKT string, or a list/character vector of geometries as WKB/WKT with length equal to the number of input geometry pairs. NULL (as_wkb = TRUE) / NA (as_wkb = FALSE) is returned with a warning if WKB input cannot be converted into an OGR geometry object, or if an error occurs in the call to the underlying OGR API function.

Note

this_geom and other_geom are assumed to be in the same coordinate reference system.

Geometry validity is not checked. In case you are unsure of the validity of the input geometries, call g_is_valid() before, otherwise the result might be wrong.

Examples

```
elev_file <- system.file("extdata/storml_elev.tif", package="gdalraster")
ds <- new(GDALRaster, elev_file)
g1 <- ds$bbox() |> bbox_to_wkt()
ds$close()

g2 <- "POLYGON ((327381.9 5104541.2, 326824.0 5104092.5, 326708.8 5103182.9,
```

```
  327885.2 5102612.9, 329334.5 5103322.4, 329304.2 5104474.5,328212.7
  5104656.4, 328212.7 5104656.4, 327381.9 5104541.2))"

# see spatial predicate defintions at https://en.wikipedia.org/wiki/DE-9IM
g_intersects(g1, g2)  # TRUE
g_overlaps(g1, g2)  # TRUE
# therefore,
g_contains(g1, g2)  # FALSE

g_sym_difference(g1, g2) |> g_area()

g3 <- g_intersection(g1, g2)
g4 <- g_union(g1, g2)
g_difference(g4, g3) |> g_area()
```

`g_binary_pred` *Geometry binary predicates operating on WKB or WKT*

Description

These functions implement tests for pairs of geometries in OGC WKB or WKT format.

Usage

```
g_intersects(this_geom, other_geom, quiet = FALSE)

g_disjoint(this_geom, other_geom, quiet = FALSE)

g_touches(this_geom, other_geom, quiet = FALSE)

g_contains(this_geom, other_geom, quiet = FALSE)

g_within(this_geom, other_geom, quiet = FALSE)

g_crosses(this_geom, other_geom, quiet = FALSE)

g_overlaps(this_geom, other_geom, quiet = FALSE)

g_equals(this_geom, other_geom, quiet = FALSE)
```

Arguments

`this_geom`	Either a raw vector of WKB or list of raw vectors, or a character vector containing one or more WKT strings.
`other_geom`	Either a raw vector of WKB or list of raw vectors, or a character vector containing one or more WKT strings. Must contain the same number of geometries as `this_geom`, unless `this_geom` contains a single geometry in which case pairwise tests will be performed for one-to-many if `other_geom` contains multiple geometries (i.e., "this-to-others").

quiet Logical value, TRUE to suppress warnings. Defaults to FALSE.

Details

These functions use the GEOS library via GDAL headers.

g_intersects() tests whether two geometries intersect.

g_disjoint() tests if this geometry and the other geometry are disjoint.

g_touches() tests if this geometry and the other geometry are touching.

g_contains() tests if this geometry contains the other geometry.

g_within() tests if this geometry is within the other geometry.

g_crosses() tests if this geometry and the other geometry are crossing.

g_overlaps() tests if this geometry and the other geometry overlap, that is, their intersection has a non-zero area (they have some but not all points in common).

g_equals() tests whether two geometries are equivalent. The GDAL documentation says: "This operation implements the SQL/MM ST_OrderingEquals() operation. The comparison is done in a structural way, that is to say that the geometry types must be identical, as well as the number and ordering of sub-geometries and vertices. Or equivalently, two geometries are considered equal by this method if their WKT/WKB representation is equal. Note: this must be distinguished from equality in a spatial way."

Value

Logical vector with length equal to the number of input geometry pairs.

Note

this_geom and other_geom are assumed to be in the same coordinate reference system.

If this_geomis a single geometry and other_geom is a list or vector of multiple geometries, then this_geom will be tested against each geometry in other_geom (otherwise no recycling is done).

Geometry validity is not checked. In case you are unsure of the validity of the input geometries, call g_is_valid() before, otherwise the result might be wrong.

See Also

https://en.wikipedia.org/wiki/DE-9IM

---

g_coords *Extract coordinate values from geometries*

Description

g_coords() extracts coordinate values (vertices) from the input geometries and returns a data frame with coordinates as columns.

Usage

```
g_coords(geom)
```

Arguments

geom	Either a raw vector of WKB or list of raw vectors, or a character vector containing one or more WKT strings.

Value

A data frame as returned by `wk::wk_coords()`: columns `feature_id` (the index of the feature from the input), `part_id` (an arbitrary integer identifying the point, line, or polygon from whence it came), `ring_id` (an arbitrary integer identifying individual rings within polygons), and one column per coordinate (`x`, `y`, and/or `z` and/or `m`).

Examples

```
dsn <- system.file("extdata/ynp_fires_1984_2022.gpkg", package="gdalraster")
lyr <- new(GDALVector, dsn)
d <- lyr$fetch(10)

vertices <- g_coords(d$geom)
head(vertices)

lyr$close()
```

---

g_envelope *Obtain the 2D or 3D bounding envelope for input geometries*

Description

`g_envelope()` computes and returns the bounding envelope(s) for the input geometries. Wrapper of `OGR_G_GetEnvelope()` / `OGR_G_GetEnvelope3D()` in the GDAL Geometry API.

Usage

```
g_envelope(geom, as_3d = FALSE, quiet = FALSE)
```

Arguments

geom	Either a raw vector of WKB or list of raw vectors, or a character vector containing one or more WKT strings.
as_3d	Logical value. `TRUE` to return the 3D bounding envelope. The 2D envelope is returned by default (`as_3d = FALSE`).
quiet	Logical value, `TRUE` to suppress warnings. Defaults to `FALSE`.

Value

Either a numeric vector of length 4 containing the 2D envelope (`xmin, xmax, ymin, ymax`) or of length 6 containing the 3D envelope (`xmin, xmax, ymin, ymax, zmin, zmax`), or a four-column or six-column numeric matrix with number of rows equal to the number of input geometries and column names (`"xmin", "xmax", "ymin", "ymax"`), or (`"xmin", "xmax", "ymin", "ymax", "zmin", "zmax"`) for the 3D case.

---

g_factory *Create WKB/WKT geometries from vertices, and add/get sub-geometries*

Description

These functions create WKB/WKT geometries from input vertices, and build container geometry types from sub-geometries.

Usage

```
g_create(
  geom_type,
  pts = NULL,
  as_wkb = TRUE,
  as_iso = FALSE,
  byte_order = "LSB"
)

g_add_geom(
  sub_geom,
  container,
  as_wkb = TRUE,
  as_iso = FALSE,
  byte_order = "LSB"
)

g_get_geom(
  container,
  sub_geom_idx,
  as_wkb = TRUE,
  as_iso = FALSE,
  byte_order = "LSB"
)
```

Arguments

geom_type	Character string (case-insensitive), one of "POINT", "MULTIPOINT", "LINESTRING", "POLYGON" (see Note) or "GEOMETRYCOLLECTION".

pts	Numeric matrix of points (x, y, z, m), or NULL to create an empty geometry. The points can be given as (x, y), (x, y, z) or (x, y, z, m), so the input must have two, three or four columns. Data frame input will be coerced to numeric matrix. Rings for polygon geometries should be closed.
as_wkb	Logical value, TRUE to return the output geometry in WKB format (the default), or FALSE to return a WKT string.
as_iso	Logical value, TRUE to export as ISO WKB/WKT (ISO 13249 SQL/MM Part 3), or FALSE (the default) to export as "Extended WKB/WKT".
byte_order	Character string specifying the byte order when output is WKB. One of "LSB" (the default) or "MSB" (uncommon).
sub_geom	Either a raw vector of WKB or a character string of WKT.
container	Either a raw vector of WKB or a character string of WKT for a container geometry type.
sub_geom_idx	An integer value giving the 1-based index of a sub-geometry (numeric values will be coerced to integer by truncation).

Details

These functions use the GEOS library via GDAL headers.

g_create() creates a geometry object from the given point(s) and returns a raw vector of WKB (the default) or a character string of WKT. Currently supports creating Point, MultiPoint, LineString, Polygon, and GeometryCollection. If multiple input points are given for creating Point type, then multiple geometries will be returned as a list of WKB raw vectors, or character vector of WKT strings (if as_wkb = FALSE). Otherwise, a single geometry is created from the input points. Only an empty GeometryCollection can be created with this function, for subsequent use with g_add_geom().

g_add_geom() adds a geometry to a geometry container, e.g., Polygon to Polygon (to add an interior ring), Point to MultiPoint, LineString to MultiLineString, Polygon to MultiPolygon, or mixed geometry types to a GeometryCollection. Returns a new geometry, i.e, the container geometry is not modified.

g_get_geom() fetches a geometry from a geometry container (1-based indexing). For a polygon, requesting the first sub-geometry returns the exterior ring (sub_geom_idx = 1), and the interior rings are returned for sub_geom_idx > 1.

Value

A geometry as WKB raw vector by default, or a WKT string if as_wkb = FALSE. In the case of multiple input points for creating Point geometry type, a list of WKB raw vectors or character vector of WKT strings will be returned.

Note

A POLYGON can be created for a single ring which will be the exterior ring. Additional POLYGONs can be created and added to an existing POLYGON with g_add_geom(). These will become interior rings. Alternatively, an empty polygon can be created with g_create("POLYGON"), followed by creation and addition of POLYGONs as sub-geometries. In that case, the first added POLYGON will be the exterior ring. The next ones will be the interior rings.

Only an empty GeometryCollection can be created with g_create(), which can then be used as a container with g_add_geom(). If given, input points will be ignored by g_create() if geom_type = "GEOMETRYCOLLECTION".

Examples

```
# raw vector of WKB by default
g_create("POINT", c(1, 2))

# as WKT
g_create("POINT", c(1, 2), as_wkb = FALSE)

# or convert in either direction
g_create("POINT", c(1, 2)) |> g_wk2wk()
g_create("POINT", c(1, 2), as_wkb = FALSE) |> g_wk2wk()

# create MultiPoint from a matrix of xyz points
x <- c(9, 1)
y <- c(1, 9)
z <- c(0, 10)
pts <- cbind(x, y, z)
mp <- g_create("MULTIPOINT", pts)
g_wk2wk(mp)
g_wk2wk(mp, as_iso = TRUE)

# create an empty container and add sub-geometries
mp2 <- g_create("MULTIPOINT")
mp2 <- g_create("POINT", c(11, 2)) |> g_add_geom(mp2)
mp2 <- g_create("POINT", c(12, 3)) |> g_add_geom(mp2)
g_wk2wk(mp2)

# get sub-geometry from container
g_get_geom(mp2, 2, as_wkb = FALSE)

# plot WKT strings or a list of WKB raw vectors
pts <- c(0, 0,
         3, 0,
         3, 4,
         0, 0)
m <- matrix(pts, ncol = 2, byrow = TRUE)
(g <- g_create("POLYGON", m, as_wkb = FALSE))
plot_geom(g)
```

---

g_measures *Compute measurements for WKB/WKT geometries*

Description

These functions compute measurements for geometries. The input geometries may be given as a single raw vector of WKB, a list of WKB raw vectors, or a character vector containing one or more WKT strings.

Usage

```
g_area(geom, quiet = FALSE)

g_centroid(geom, quiet = FALSE)

g_distance(geom, other_geom, quiet = FALSE)

g_length(geom, quiet = FALSE)

g_geodesic_area(geom, srs, traditional_gis_order = TRUE, quiet = FALSE)

g_geodesic_length(geom, srs, traditional_gis_order = TRUE, quiet = FALSE)
```

Arguments

geom	Either a raw vector of WKB or list of raw vectors, or a character vector containing one or more WKT strings.
quiet	Logical value, TRUE to suppress warnings. Defaults to FALSE.
other_geom	Either a raw vector of WKB or list of raw vectors, or a character vector containing one or more WKT strings. Must contain the same number of geometries as geom, unless geom contains a single geometry in which case pairwise distances will be computed for one-to-many if other_geom contains multiple geometries (i.e., "this-to-others").
srs	Character string specifying the spatial reference system for geom. May be in WKT format or any of the formats supported by srs_to_wkt().
traditional_gis_order	Logical value, TRUE to use traditional GIS order of axis mapping (the default) or FALSE to use authority compliant axis order. By default, input geom vertices are assumed to be in longitude/latitude order if srs is a geographic coordinate system. This can be overridden by setting traditional_gis_order = FALSE.

Details

These functions use the GEOS library via GDAL headers.

g_area() computes the area for a Polygon or MultiPolygon. Undefined for all other geometry types (returns zero). Returns a numeric vector, having length equal to the number of input geometries, containing computed area or '0' if undefined.

g_centroid() returns a numeric vector of length 2 containing the centroid (X, Y), or a two-column numeric matrix (X, Y) with number of rows equal to the number of input geometries. The GDAL documentation states "This method relates to the SFCOM ISurface::get_Centroid() method however the current implementation based on GEOS can operate on other geometry types such as multipoint, linestring, geometrycollection such as multipolygons. OGC SF SQL 1.1 defines the operation for surfaces (polygons). SQL/MM-Part 3 defines the operation for surfaces and multisurfaces (multipolygons)."

g_distance() returns the distance between two geometries or -1 if an error occurs. Returns the shortest distance between the two geometries. The distance is expressed into the same unit as the

coordinates of the geometries. Returns a numeric vector, having length equal to the number of input geometry pairs, containing computed distance or '-1' if an error occurs.

`g_length()` computes the length for `LineString` or `MultiCurve` objects. Undefined for all other geometry types (returns zero). Returns a numeric vector, having length equal to the number of input geometries, containing computed length or '0' if undefined.

`g_geodesic_area()` computes geometry area, considered as a surface on the underlying ellipsoid of the SRS attached to the geometry. The returned area will always be in square meters, and assumes that polygon edges describe geodesic lines on the ellipsoid. If the geometry SRS is not a geographic one, geometries are reprojected to the underlying geographic SRS. By default, input geometry vertices are assumed to be in longitude/latitude order if using a geographic coordinate system. This can be overridden with the `traditional_gis_order` argument. Returns the area in square meters, or `NA` in case of error (unsupported geometry type, no SRS attached, etc.) Requires GDAL >= 3.9.

`g_geodesic_length()` computes the length of the curve, considered as a geodesic line on the underlying ellipsoid of the SRS attached to the geometry. The returned length will always be in meters. If the geometry SRS is not a geographic one, geometries are reprojected to the underlying geographic SRS. By default, input geometry vertices are assumed to be in longitude/latitude order if using a geographic coordinate system. This can be overridden with the `traditional_gis_order` argument. Returns the length in meters, or `NA` in case of error (unsupported geometry type, no SRS attached, etc.) Requires GDAL >= 3.10.

Note

For `g_distance()`, `geom` and `other_geom` must be in the same coordinate reference system. If `geom` is a single geometry and `other_geom` is a list or vector of multiple geometries, then distances will be calculated between `geom` and each geometry in `other_geom`. Otherwise, no recycling is done and `length(geom)` must equal `length(other_geom)` to calculate distance between each corresponding pair of input geometries.

Geometry validity is not checked. In case you are unsure of the validity of the input geometries, call `g_is_valid()` before, otherwise the result might be wrong.

Examples

```
g_area("POLYGON ((0 0, 10 10, 10 0, 0 0))")

g_centroid("POLYGON ((0 0, 10 10, 10 0, 0 0))")

g_distance("POINT (0 0)", "POINT (5 12)")

g_length("LINESTRING (0 0, 3 4)")

f <- system.file("extdata/ynp_fires_1984_2022.gpkg", package = "gdalraster")
lyr <- new(GDALVector, f, "mtbs_perims")

# read all features into a data frame
feat_set <- lyr$fetch(-1)
head(feat_set)

g_area(feat_set$geom) |> head()
```

```
g_centroid(feat_set$geom) |> head()

lyr$close()
```

---

g_query *Obtain information about WKB/WKT geometries*

Description

These functions return information about WKB/WKT geometries. The input geometries may be given as a single raw vector of WKB, a list of WKB raw vectors, or a character vector containing one or more WKT strings.

Usage

```
g_is_empty(geom, quiet = FALSE)

g_is_valid(geom, quiet = FALSE)

g_is_3D(geom, quiet = FALSE)

g_is_measured(geom, quiet = FALSE)

g_is_ring(geom, quiet = FALSE)

g_name(geom, quiet = FALSE)

g_summary(geom, quiet = FALSE)

g_geom_count(geom, quiet = FALSE)
```

Arguments

geom	Either a raw vector of WKB or list of raw vectors, or a character vector containing one or more WKT strings.
quiet	Logical value, TRUE to suppress warnings. Defaults to FALSE.

Details

g_is_empty() tests whether a geometry has no points. Returns a logical vector of the same length as the number of input geometries containing TRUE for the corresponding geometries that are empty or FALSE for non-empty geometries.

g_is_valid() tests whether a geometry is valid. Returns a logical vector analogous to the above for g_is_empty().

g_is_3D() checks whether a geometry has Z coordinates. Returns a logical vector analogous to the above for g_is_empty().

g_is_measured() checks whether a geometry is measured (has M values). Returns a logical vector analogous to the above for g_is_empty().

g_is_ring() tests whether a geometry is a ring, TRUE if the coordinates of the geometry form a ring by checking length and closure (self-intersection is not checked), otherwise FALSE. Returns a logical vector analogous to the above for g_is_empty().

g_name() returns the WKT type names of the input geometries in a character vector of the same length as the number of input geometries.

g_summary() returns text summaries of WKB/WKT geometries in a character vector of the same length as the number of input geometries. Requires GDAL >= 3.7.

g_geom_count() returns the number of elements in a geometry or number of geometries in container. Only geometries of type Polygon[25D], MultiPoint[25D], MultiLineString[25D], MultiPolygon[25D] or GeometryCollection[25D] may return a valid value. Other geometry types will silently return 0.

See Also

g_make_valid(), g_set_3D(), g_set_measured()

Examples

```
g1 <- "POLYGON ((0 0, 10 10, 10 0, 0 0))"
g2 <- "POLYGON ((5 1, 9 5, 9 1, 5 1))"
g_difference(g2, g1) |> g_is_empty()

g1 <- "POLYGON ((0 0, 10 10, 10 0, 0 0))"
g2 <- "POLYGON ((0 0, 10 10, 10 0))"
g3 <- "POLYGON ((0 0, 10 10, 10 0, 0 1))"
g_is_valid(c(g1, g2, g3))

g_is_3D(g1)
g_is_measured(g1)

pt_xyz <- g_create("POINT", c(1, 9, 100))
g_is_3D(pt_xyz)
g_is_measured(pt_xyz)

pt_xyzm <- g_create("POINT", c(1, 9, 100, 2000))
g_is_3D(pt_xyzm)
g_is_measured(pt_xyzm)

f <- system.file("extdata/ynp_fires_1984_2022.gpkg", package = "gdalraster")
lyr <- new(GDALVector, f, "mtbs_perims")

feat <- lyr$getNextFeature()
g_name(feat$geom)

# g_summary() requires GDAL >= 3.7
if (gdal_version_num() >= gdal_compute_version(3, 7, 0)) {
  feat <- lyr$getNextFeature()
  g_summary(feat$geom) |> print()
```

```
  feat_set <- lyr$fetch(5)
  g_summary(feat_set$geom) |> print()
}

g_geom_count(feat$geom)

lyr$close()
```

---

g_transform *Apply a coordinate transformation to a WKB/WKT geometry*

Description

g_transform() will transform the coordinates of a geometry from their current spatial reference system to a new target spatial reference system. Normally this means reprojecting the vectors, but it could include datum shifts, and changes of units.

Usage

```
g_transform(
  geom,
  srs_from,
  srs_to,
  wrap_date_line = FALSE,
  date_line_offset = 10L,
  traditional_gis_order = TRUE,
  as_wkb = TRUE,
  as_iso = FALSE,
  byte_order = "LSB",
  quiet = FALSE
)
```

Arguments

geom	Either a raw vector of WKB or list of raw vectors, or a character vector containing one or more WKT strings.
srs_from	Character string specifying the spatial reference system for geom. May be in WKT format or any of the formats supported by srs_to_wkt().
srs_to	Character string specifying the output spatial reference system. May be in WKT format or any of the formats supported by srs_to_wkt().
wrap_date_line	Logical value, TRUE to correct geometries that incorrectly go from a longitude on a side of the antimeridian to the other side. Defaults to FALSE.
date_line_offset	Integer longitude gap in degree. Defaults to 10L.

traditional_gis_order
: Logical value, TRUE to use traditional GIS order of axis mapping (the default) or FALSE to use authority compliant axis order. By default, input geom vertices are assumed to be in longitude/latitude order if srs_from is a geographic coordinate system. This can be overridden by setting traditional_gis_order = FALSE.

as_wkb
: Logical value, TRUE to return the output geometry in WKB format (the default), or FALSE to return as WKT.

as_iso
: Logical value, TRUE to export as ISO WKB/WKT (ISO 13249 SQL/MM Part 3), or FALSE (the default) to export as "Extended WKB/WKT".

byte_order
: Character string specifying the byte order when output is WKB. One of "LSB" (the default) or "MSB" (uncommon).

quiet
: Logical value, TRUE to suppress warnings. Defaults to FALSE.

Value

A geometry as WKB raw vector or WKT string, or a list/character vector of geometries as WKB/WKT with length equal to the number of input geometries. NULL (as_wkb = TRUE) / NA (as_wkb = FALSE) is returned with a warning if WKB input cannot be converted into an OGR geometry object, or if an error occurs in the call to the underlying OGR API.

Note

This function uses the OGR_GeomTransformer_Create() and OGR_GeomTransformer_Transform() functions in the GDAL API: "This is an enhanced version of OGR_G_Transform(). When reprojecting geometries from a Polar Stereographic projection or a projection naturally crossing the antimeridian (like UTM Zone 60) to a geographic CRS, it will cut geometries along the antimeridian. So a LineString might be returned as a MultiLineString."

The wrap_date_line = TRUE option might be specified for circumstances to correct geometries that incorrectly go from a longitude on a side of the antimeridian to the other side, e.g., LINESTRING (-179 0,179 0) will be transformed to MULTILINESTRING ((-179 0,-180 0),(180 0,179 0)). For that use case, srs_to might be the same as srs_from.

See Also

bbox_transform(), transform_bounds()

Examples

```
pt <- "POINT (-114.0 47.0)"
g_transform(pt, "WGS84", "EPSG:5070", as_wkb = FALSE)

# correct geometries that incorrectly go from a longitude on a side of the
# antimeridian to the other side
geom <- "LINESTRING (-179 0,179 0)"
g_transform(geom, "WGS84", "WGS84", wrap_date_line = TRUE, as_wkb = FALSE)
```

g_unary_op *Unary operations on WKB or WKT geometries*

Description

These functions implement algorithms that operate on one input geometry for which a new output geometry is generated. The input geometries may be given as a single raw vector of WKB, a list of WKB raw vectors, or a character vector containing one or more WKT strings.

Usage

```
g_buffer(
  geom,
  dist,
  quad_segs = 30L,
  as_wkb = TRUE,
  as_iso = FALSE,
  byte_order = "LSB",
  quiet = FALSE
)

g_boundary(
  geom,
  as_wkb = TRUE,
  as_iso = FALSE,
  byte_order = "LSB",
  quiet = FALSE
)

g_convex_hull(
  geom,
  as_wkb = TRUE,
  as_iso = FALSE,
  byte_order = "LSB",
  quiet = FALSE
)

g_concave_hull(
  geom,
  ratio,
  allow_holes,
  as_wkb = TRUE,
  as_iso = FALSE,
  byte_order = "LSB",
  quiet = FALSE
)
```

```
g_delaunay_triangulation(
  geom,
  constrained = FALSE,
  tolerance = 0,
  only_edges = FALSE,
  as_wkb = TRUE,
  as_iso = FALSE,
  byte_order = "LSB",
  quiet = FALSE
)

g_simplify(
  geom,
  tolerance,
  preserve_topology = TRUE,
  as_wkb = TRUE,
  as_iso = FALSE,
  byte_order = "LSB",
  quiet = FALSE
)

g_unary_union(
  geom,
  as_wkb = TRUE,
  as_iso = FALSE,
  byte_order = "LSB",
  quiet = FALSE
)
```

Arguments

geom	Either a raw vector of WKB or list of raw vectors, or a character vector containing one or more WKT strings.
dist	Numeric buffer distance in units of the input geom.
quad_segs	Integer number of segments used to define a 90 degree curve (quadrant of a circle). Large values result in large numbers of vertices in the resulting buffer geometry while small numbers reduce the accuracy of the result.
as_wkb	Logical value, TRUE to return the output geometry in WKB format (the default), or FALSE to return as WKT.
as_iso	Logical value, TRUE to export as ISO WKB/WKT (ISO 13249 SQL/MM Part 3), or FALSE (the default) to export as "Extended WKB/WKT".
byte_order	Character string specifying the byte order when output is WKB. One of "LSB" (the default) or "MSB" (uncommon).
quiet	Logical value, TRUE to suppress warnings. Defaults to FALSE.
ratio	Numeric value in interval [0, 1]. The target criterion parameter for g_concave_hull(), expressed as a ratio between the lengths of the longest and shortest edges. 1 produces the convex hull; 0 produces a hull with maximum concaveness (see Note).

allow_holes — Logical value, whether holes are allowed.

constrained — Logical value, TRUE to return a constrained Delaunay triangulation of the vertices of the given polygon(s). Defaults to FALSE.

tolerance — Numeric value. For g_simplify(), the simplification tolerance as distance in units of the input geom. Simplification removes vertices which are within the tolerance distance of the simplified linework (as long as topology is preserved when preserve_topology = TRUE). For g_delaunay_triangulation(), an optional snapping tolerance to use for improved robustness (ignored if constrained = TRUE).

only_edges — Logical value. If TRUE, g_delaunay_triangulation() will return a MULTILINESTRING, otherwise it will return a GEOMETRYCOLLECTION containing triangular POLYGONs (the default). Ignored if constrained = TRUE

preserve_topology — Logical value, TRUE to simplify geometries while preserving topology (the default). Setting to FALSE simplifies geometries using the standard Douglas-Peucker algorithm which is significantly faster (see Note).

Details

These functions use the GEOS library via GDAL headers.

g_boundary() computes the boundary of a geometry. Wrapper of OGR_G_Boundary() in the GDAL Geometry API.

g_buffer() builds a new geometry containing the buffer region around the geometry on which it is invoked. The buffer is a polygon containing the region within the buffer distance of the original geometry. Wrapper of OGR_G_Buffer() in the GDAL API.

g_convex_hull() computes a convex hull, the smallest convex geometry that contains all the points in the input geometry. Wrapper of OGR_G_ConvexHull() in the GDAL API.

g_concave_hull() returns a "concave hull" of a geometry. A concave hull is a polygon which contains all the points of the input, but is a better approximation than the convex hull to the area occupied by the input. Frequently used to convert a multi-point into a polygonal area that contains all the points in the input geometry. Requires GDAL >= 3.6 and GEOS >= 3.11.

g_delaunay_triangulation()

- constrained = FALSE: returns a Delaunay triangulation of the vertices of the input geometry. Wrapper of OGR_G_DelaunayTriangulation() in the GDAL API. Requires GEOS >= 3.4.
- constrained = TRUE: returns a constrained Delaunay triangulation of the vertices of the given polygon(s). For non-polygonal inputs, silently returns an empty geometry collection. Requires GDAL >= 3.12 and GEOS >= 3.10.

g_simplify() computes a simplified geometry. By default, it simplifies the input geometries while preserving topology (see Note). Wrapper of OGR_G_Simplify() / OGR_G_SimplifyPreserveTopology() in the GDAL API.

g_unary_union() returns the union of all components of a single geometry. Usually used to convert a collection into the smallest set of polygons that cover the same area. See https://postgis.net/docs/ST_UnaryUnion.html for more details. Requires GDAL >= 3.7.

Value

A geometry as WKB raw vector or WKT string, or a list/character vector of geometries as WKB/WKT with length equal to the number of input geometries. NULL (as_wkb = TRUE) / NA (as_wkb = FALSE) is returned with a warning if WKB input cannot be converted into an OGR geometry object, or if an error occurs in the call to the underlying OGR API.

Note

Definitions of these operations are given in the GEOS documentation (https://libgeos.org/doxygen/, GEOS 3.15.0dev), some of which is copied here.

g_boundary() computes the "boundary" as defined by the DE9IM (https://en.wikipedia.org/wiki/DE-9IM):

- the boundary of a Polygon is the set of linear rings dividing the exterior from the interior
- the boundary of a LineString is the two end points
- the boundary of a Point/MultiPoint is defined as empty

g_buffer() always returns a polygonal result. The negative or zero-distance buffer of lines and points is always an empty Polygon.

g_convex_hull() uses the Graham Scan algorithm.

g_concave_hull(): A set of points has a sequence of hulls of increasing concaveness, determined by a numeric target parameter. The concave hull is constructed by removing the longest outer edges of the Delaunay Triangulation of the space between the polygons, until the target criterion parameter is reached. This can be expressed as a ratio between the lengths of the longes and shortest edges. 1 produces the convex hull; 0 produces a hull with maximum concaveness.

g_simplify():

- With preserve_topology = TRUE (the default):
  Simplifies a geometry, ensuring that the result is a valid geometry having the same dimension and number of components as the input. The simplification uses a maximum distance difference algorithm similar to the one used in the Douglas-Peucker algorithm. In particular, if the input is an areal geometry (Polygon or MultiPolygon), the result has the same number of shells and holes (rings) as the input, in the same order. The result rings touch at no more than the number of touching point in the input (although they may touch at fewer points).
- With preserve_topology = FALSE:
  Simplifies a geometry using the standard Douglas-Peucker algorithm. Ensures that any polygonal geometries returned are valid. Simple lines are not guaranteed to remain simple after simplification. Note that in general D-P does not preserve topology - e.g. polygons can be split, collapse to lines or disappear, holes can be created or disappear, and lines can cross. To simplify geometry while preserving topology use TopologyPreservingSimplifier. (However, using D-P is significantly faster.)

preserve_topology = TRUE does not preserve boundaries shared between polygons.

Examples

```
g <- "POLYGON((0 0,1 1,1 0,0 0))"
g_boundary(g, as_wkb = FALSE)

g <- "POINT (0 0)"
g_buffer(g, dist = 10, as_wkb = FALSE)

g <- "GEOMETRYCOLLECTION(POINT(0 1), POINT(0 0), POINT(1 0), POINT(1 1))"
g_convex_hull(g, as_wkb = FALSE)

# g_concave_hull() requires GDAL >= 3.6 and GEOS >= 3.11
if (gdal_version_num() >= gdal_compute_version(3, 6, 0) &&
    (geos_version()$major > 3 || geos_version()$minor >= 11)) {
  g <- "MULTIPOINT(0 0,0.4 0.5,0 1,1 1,0.6 0.5,1 0)"
  g_concave_hull(g, ratio = 0.5, allow_holes = FALSE, as_wkb = FALSE)
}

# g_delaunay_triangulation() requires GEOS >= 3.4
if (geos_version()$major > 3 || geos_version()$minor >= 4) {
  g <- "MULTIPOINT(0 0,0 1,1 1,1 0)"
  g_delaunay_triangulation(g, as_wkb = FALSE)
}

g <- "LINESTRING(0 0,1 1,10 0)"
g_simplify(g, tolerance = 5, as_wkb = FALSE)

# g_unary_union() requires GDAL >= 3.7
if (gdal_version_num() >= gdal_compute_version(3, 7, 0)) {
  g <- "GEOMETRYCOLLECTION(POINT(0.5 0.5), POLYGON((0 0,0 1,1 1,1 0,0 0)),
        POLYGON((1 0,1 1,2 1,2 0,1 0)))"
  g_unary_union(g, as_wkb = FALSE)
}
```

---

g_util *Geometry utility functions operating on WKB or WKT*

Description

These functions operate on input geometries in OGC WKB or WKT format to perform various manipulations for utility purposes.

Usage

```
g_make_valid(
  geom,
  method = "LINEWORK",
  keep_collapsed = FALSE,
  as_wkb = TRUE,
  as_iso = FALSE,
```

```
  byte_order = "LSB",
  quiet = FALSE
)

g_normalize(
  geom,
  as_wkb = TRUE,
  as_iso = FALSE,
  byte_order = "LSB",
  quiet = FALSE
)

g_set_3D(
  geom,
  is_3d,
  as_wkb = TRUE,
  as_iso = FALSE,
  byte_order = "LSB",
  quiet = FALSE
)

g_set_measured(
  geom,
  is_measured,
  as_wkb = TRUE,
  as_iso = FALSE,
  byte_order = "LSB",
  quiet = FALSE
)

g_swap_xy(
  geom,
  as_wkb = TRUE,
  as_iso = FALSE,
  byte_order = "LSB",
  quiet = FALSE
)
```

Arguments

geom	Either a raw vector of WKB or list of raw vectors, or a character vector containing one or more WKT strings.
method	Character string. One of "LINEWORK" (the default) or "STRUCTURE" (requires GEOS >= 3.10 and GDAL >= 3.4). See Details.
keep_collapsed	Logical value, applies only to the STRUCTURE method. Defaults to FALSE. See Details.
as_wkb	Logical value, TRUE to return the output geometry in WKB format (the default), or FALSE to return as WKT.

as_iso	Logical value, TRUE to export as ISO WKB/WKT (ISO 13249 SQL/MM Part 3), or FALSE (the default) to export as "Extended WKB/WKT".
byte_order	Character string specifying the byte order when output is WKB. One of "LSB" (the default) or "MSB" (uncommon).
quiet	Logical value, TRUE to suppress warnings. Defaults to FALSE.
is_3d	Logical value, TRUE if the input geometries should have a Z dimension, or FALSE to remove the Z dimension.
is_measured	Logical value, TRUE if the input geometries should have a M dimension, or FALSE to remove the M dimension.

Details

These functions use the GEOS library via GDAL headers.

g_make_valid() attempts to make an invalid geometry valid without losing vertices. Already-valid geometries are cloned without further intervention. Wrapper of OGR_G_MakeValid()/OGR_G_MakeValidEx() in the GDAL API. Requires the GEOS >= 3.8 library, check it for the definition of the geometry operation. If GDAL is built without GEOS >= 3.8, this function will return a clone of the input geometry if it is valid, or NULL (as_wkb = TRUE) / NA (as_wkb = FALSE) if it is invalid.

- "LINEWORK" is the default method, which combines all rings into a set of noded lines and then extracts valid polygons from that linework (requires GEOS >= 3.10 and GDAL >= 3.4). The "STRUCTURE" method first makes all rings valid, then merges shells and subtracts holes from shells to generate a valid result. Assumes that holes and shells are correctly categorized.
- keep_collapsed only applies to the "STRUCTURE" method:
  - FALSE (the default): collapses are converted to empty geometries
  - TRUE: collapses are converted to a valid geometry of lower dimension

g_normalize() organizes the elements, rings, and coordinate order of geometries in a consistent way, so that geometries that represent the same object can be easily compared. Wrapper of OGR_G_Normalize() in the GDAL API. Requires GDAL >= 3.3. Normalization ensures the following:

- Lines are oriented to have smallest coordinate first (apart from duplicate endpoints)
- Rings start with their smallest coordinate (using XY ordering)
- Polygon shell rings are oriented clockwise, and holes counter-clockwise
- Collection elements are sorted by their first coordinate

g_set_3D() adds or removes the explicit Z coordinate dimension. Removing the Z coordinate dimension of a geometry will remove any existing Z values. Adding the Z dimension to a geometry collection, a compound curve, a polygon, etc. will affect the children geometries. Wrapper of OGR_G_Set3D() in the GDAL API.

g_set_measured() adds or removes the explicit M coordinate dimension. Removing the M coordinate dimension of a geometry will remove any existing M values. Adding the M dimension to a geometry collection, a compound curve, a polygon, etc. will affect the children geometries. Wrapper of OGR_G_SetMeasured() in the GDAL API.

g_swap_xy() swaps x and y coordinates of the input geometry. Wrapper of OGR_G_SwapXY() in the GDAL API.

Value

A geometry as WKB raw vector or WKT string, or a list/character vector of geometries as WKB/WKT with length equal to `length(geom)`. `NULL` is returned with a warning if WKB input cannot be converted into an OGR geometry object, or if an error occurs in the call to the underlying OGR API.

See Also

`g_is_valid()`, `g_is_3D()`, `g_is_measured()`

Examples

```
## g_make_valid() requires GEOS >= 3.8, otherwise is only a validity test
geos_version()

# valid
wkt <- "POINT (0 0)"
g_make_valid(wkt, as_wkb = FALSE)

# invalid to valid
wkt <- "POLYGON ((0 0,10 10,0 10,10 0,0 0))"
g_make_valid(wkt, as_wkb = FALSE)

# invalid - error
wkt <- "LINESTRING (0 0)"
g_make_valid(wkt)  # NULL

## g_normalize() requires GDAL >= 3.3
if (gdal_version_num() >= gdal_compute_version(3, 3, 0)) {
  g <- "POLYGON ((0 1,1 1,1 0,0 0,0 1))"
  g_normalize(g) |> g_wk2wk()
}

## set 3D / set measured
pt_xyzm <- g_create("POINT", c(1, 9, 100, 2000))

g_wk2wk(pt_xyzm, as_iso = TRUE)

g_set_3D(pt_xyzm, is_3d = FALSE) |> g_wk2wk(as_iso = TRUE)

g_set_measured(pt_xyzm, is_measured = FALSE) |> g_wk2wk(as_iso = TRUE)

## swap XY
g <- "GEOMETRYCOLLECTION(POINT(1 2),
                         LINESTRING(1 2,2 3),
                         POLYGON((0 0,0 1,1 1,0 0)))"

g_swap_xy(g, as_wkb = FALSE)
```

g_wk2wk *Geometry WKB/WKT conversion*

Description

g_wk2wk() converts geometries between Well Known Binary (WKB) and Well Known Text (WKT) formats. A geometry given as a raw vector of WKB will be converted to a WKT string, while a geometry given as a WKT string will be converted to a WKB raw vector. Input may also be a list of WKB raw vectors or a character vector of WKT strings.

Usage

```
g_wk2wk(geom, as_iso = FALSE, byte_order = "LSB")
```

Arguments

geom	Either a raw vector of WKB or list of raw vectors to convert to WKT, or a character vector containing one or more WKT strings to convert to WKB.
as_iso	Logical value, TRUE to export as ISO WKB/WKT (ISO 13249 SQL/MM Part 3), or FALSE (the default) to export as "Extended WKB/WKT" (see Note).
byte_order	Character string specifying the byte order when converting to WKB. One of "LSB" (the default) or "MSB" (uncommon).

Value

For input of a WKB raw vector or list of raw vectors, returns a character vector of WKT strings, with length of the returned vector equal to the number of input raw vectors. For input of a single WKT string, returns a raw vector of WKB. For input of a character vector containing more than one WKT string, returns a list of WKB raw vectors, with length of the returned list equal to the number of input strings.

Note

With as_iso = FALSE (the default), geometries are exported as extended dimension (Z) WKB/WKT for types Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon and GeometryCollection. For other geometry types, it is equivalent to ISO.

When the return value is a list of WKB raw vectors, an element in the returned list will contain NULL (and a warning emitted) if the corresponding input string was NA or empty ("").

When input is a list of WKB raw vectors, a corresponding element in the returned character vector will be NA if the input was a raw vector of length 0 (i.e., raw(0)). If an input list element is not a raw vector, then the corresponding element in the returned character vector will also be NA. A warning is emitted in each case.

See Also

GEOS reference for geometry formats:
https://libgeos.org/specifications/

Examples

```
wkt <- "POINT (-114 47)"
wkb <- g_wk2wk(wkt)
str(wkb)
g_wk2wk(wkb)
```

---

has_geos *Is GEOS available?*

Description

has_geos() returns a logical value indicating whether GDAL was built against the GEOS library. GDAL built with GEOS is a system requirement as of gdalraster 1.10.0, so this function will always return TRUE (may be removed in a future version).

Usage

```
has_geos()
```

Value

Logical. TRUE if GEOS is available, otherwise FALSE.

Examples

```
has_geos()
```

---

has_spatialite *Is SpatiaLite available?*

Description

has_spatialite() returns a logical value indicating whether GDAL was built with support for the SpatiaLite library. SpatiaLite extends the SQLite core to support full Spatial SQL capabilities.

Usage

```
has_spatialite()
```

Details

GDAL supports executing SQL statements against a datasource. For most file formats (e.g. Shapefiles, GeoJSON, FlatGeobuf files), the built-in OGR SQL dialect will be used by default. It is also possible to request the alternate "SQLite" dialect, which will use the SQLite engine to evaluate commands on GDAL datasets. This assumes that GDAL is built with support for SQLite, and preferably with Spatialite support too to benefit from spatial functions.

Value

Logical scalar. TRUE if SpatiaLite is available to GDAL.

Note

All GDAL/OGR drivers for database systems, e.g., PostgreSQL / PostGIS, Oracle Spatial, SQLite / Spatialite RDBMS, GeoPackage, etc., override the GDALDataset::ExecuteSQL() function with a dedicated implementation and, by default, pass the SQL statements directly to the underlying RDBMS. In these cases the SQL syntax varies in some particulars from OGR SQL. Also, anything possible in SQL can then be accomplished for these particular databases. For those drivers, it is also possible to explicitly request the OGRSQL or SQLite dialects, although performance will generally be much less than the native SQL engine of those database systems.

See Also

ogrinfo(), ogr_execute_sql()

OGR SQL dialect and SQLITE SQL dialect:
https://gdal.org/en/stable/user/ogr_sql_sqlite_dialect.html

Examples

```
has_spatialite()
```

---

http_enabled *Check if GDAL CPLHTTP services can be useful (libcurl)*

Description

http_enabled() returns TRUE if libcurl support is enabled. Wrapper of CPLHTTPEnabled() in the GDAL Common Portability Library.

Usage

```
http_enabled()
```

Value

Logical scalar, TRUE if GDAL was built with libcurl support.

Examples

```
http_enabled()
```

identifyDriver *Identify the GDAL driver that can open a dataset*

Description

identifyDriver() will try to identify the driver that can open the passed file name by invoking the Identify method of each registered GDALDriver in turn. The short name of the first driver that successfully identifies the file name will be returned as a character string. If all drivers fail then NULL is returned. Wrapper of GDALIdentifyDriverEx() in the GDAL C API.

Usage

```
identifyDriver(
  filename,
  raster = TRUE,
  vector = TRUE,
  allowed_drivers = NULL,
  file_list = NULL
)
```

Arguments

filename	Character string containing the name of the file to access. This may not refer to a physical file, but instead contain information for the driver on how to access a dataset (e.g., connection string, URL, etc.)
raster	Logical value indicating whether to include raster format drivers in the search, TRUE by default. May be set to FALSE to include only vector drivers.
vector	Logical value indicating whether to include vector format drivers in the search, TRUE by default. May be set to FALSE to include only raster drivers.
allowed_drivers	Optional character vector of driver short names that must be considered. Set to NULL to consider all candidate drivers (the default).
file_list	Optional character vector of filenames, including those that are auxiliary to the main filename (see Note). May contain the input filename but this is not required. Defaults to NULL.

Value

A character string with the short name of the first driver that successfully identifies the input file name, or NULL on failure.

Note

In order to reduce the need for such searches to touch the file system machinery of the operating system, it is possible to give an optional list of files. This is the list of all files at the same level in

the file system as the target file, including the target file. The filenames should not include any path components. If the target object does not have filesystem semantics then the file list should be NULL.

At least one of the raster or vector arguments must be TRUE.

See Also

gdal_formats()

Examples

```
src <- system.file("extdata/ynp_fires_1984_2022.gpkg", package="gdalraster")

identifyDriver(src) |> gdal_formats()
```

---

inspectDataset *Obtain information about a GDAL raster or vector dataset*

Description

inspectDataset() returns information about the format and content of a dataset. The function first calls identifyDriver(), and then opens the dataset as raster and/or vector to obtain information about its content. The return value is a list with named elements.

Usage

```
inspectDataset(filename, ...)
```

Arguments

filename	Character string containing the name of the file to access. This may not refer to a physical file, but instead contain information for the driver on how to access a dataset (e.g., connection string, URL, etc.)
...	Additional arguments passed to identifyDriver().

Value

A list with the following named elements:

- format: character string, the format short name
- supports_raster: logical, TRUE if the format supports raster data
- contains_raster: logical, TRUE if this is a raster dataset or the source contains raster subdatasets
- supports_subdatasets: logical, TRUE if the format supports raster subdatasets
- contains_subdatasets: logical, TRUE if the source contains subdatasets
- subdataset_names: character vector containing the subdataset names, or empty vector if subdatasets are not supported or not present

- supports_vector: logical, TRUE if the format supports vector data
- contains_vector: logical, TRUE if the source contains one or more vector layers
- layer_names: character vector containing the vector layer names, or empty vector if the format does not support vector or the source does not contain any vector layers

Note

Subdataset names are the character strings that can be used to instantiate GDALRaster objects. See https://gdal.org/en/stable/en/latest/user/raster_data_model.html#subdatasets-domain.

PostgreSQL / PostGISRaster are handled as a special case. If additional arguments raster or vector are not given for identifyDriver(), then raster = FALSE is assumed.

See Also

gdal_formats(), identifyDriver()

Examples

```
f <- system.file("extdata/ynp_features.zip", package = "gdalraster")
ynp_dsn <- file.path("/vsizip", f, "ynp_features.gpkg")

inspectDataset(ynp_dsn)
```

---

inv_geotransform *Invert geotransform*

Description

inv_geotransform() inverts a vector of geotransform coefficients. This converts the equation from being:
raster pixel/line (column/row) -> geospatial x/y coordinate
to:
geospatial x/y coordinate -> raster pixel/line (column/row)

Usage

```
inv_geotransform(gt)
```

Arguments

gt	Numeric vector of length six containing the geotransform to invert.

Value

Numeric vector of length six containing the inverted geotransform. The output vector will contain NAs if the input geotransform is uninvertable.

See Also

GDALRaster$getGeoTransform(), get_pixel_line()

Examples

```
elev_file <- system.file("extdata/storml_elev.tif", package="gdalraster")
ds <- new(GDALRaster, elev_file)
invgt <- ds$getGeoTransform() |> inv_geotransform()
ds$close()

ptX = 324181.7
ptY = 5103901.4

## for a point x, y in the spatial reference system of elev_file
## raster pixel (column number):
pixel <- floor(invgt[1] +
               invgt[2] * ptX +
               invgt[3] * ptY)

## raster line (row number):
line <- floor(invgt[4] +
              invgt[5] * ptX +
              invgt[6] * ptY)

## get_pixel_line() applies this conversion
```

---

inv_project *Inverse project geospatial x/y coordinates to longitude/latitude*

Description

`inv_project()` transforms geospatial x/y coordinates to longitude/latitude in the same geographic coordinate system used by the given projected spatial reference system. The output long/lat can optionally be set to a specific geographic coordinate system by specifying a well known name (see Details).

Usage

```
inv_project(pts, srs, well_known_gcs = NULL)
```

Arguments

`pts`	A data frame or numeric matrix containing geospatial point coordinates, or point geometries as a list of WKB raw vectors or character vector of WKT strings. If data frame or matrix, the number of columns must be either two (x, y), three (x, y, z) or four (x, y, z, t). May be also be given as a numeric vector for one point (xy, xyz, or xyzt).
`srs`	Character string specifying the projected spatial reference system for `pts`. May be in WKT format or any of the formats supported by `srs_to_wkt()`.

`well_known_gcs` Optional character string containing a supported well known name of a geographic coordinate system (see Details for supported values).

Details

By default, the geographic coordinate system of the projection specified by `srs` will be used. If a specific geographic coordinate system is desired, then `well_known_gcs` can be set to one of the values below:

EPSG:n	where n is the code of a geographic coordinate system
WGS84	same as EPSG:4326
WGS72	same as EPSG:4322
NAD83	same as EPSG:4269
NAD27	same as EPSG:4267
CRS84	same as WGS84
CRS72	same as WGS72
CRS27	same as NAD27

The coordinates returned by `inv_project()`will always be in longitude, latitude order (traditional GIS order) regardless of the axis order defined for the GCS names above.

Value

Numeric matrix of longitude, latitude (potentially also with z, or z and t columns).

Note

Input points that contain missing values (`NA`) will be assigned `NA` in the output and a warning emitted. Input points that fail to transform with the GDAL API call will also be assigned `NA` in the output with a specific warning indicating that case.

See Also

`transform_xy()`

Examples

```
pt_file <- system.file("extdata/storml_pts.csv", package="gdalraster")
# id, x, y in NAD83 / UTM zone 12N
pts <- read.csv(pt_file)
print(pts)
inv_project(pts[,-1], "EPSG:26912")
```

make_chunk_index *Generate an index of chunk offsets and sizes for iterating a raster*

Description

make_chunk_index() returns a matrix of xchunkoff, ychunkoff, xoff, yoff, xsize, ysize, xmin, xmax, ymin and ymax, i.e., indexing of potentially multi-block chunks defined on block boundaries for iterating I/O operations over a raster. The last four columns are geospatial coordinates of the bounding box. Note that class GDALRaster provides a method of the same name that is more convenient to use with a dataset object.

Usage

```
make_chunk_index(
  raster_xsize,
  raster_ysize,
  block_xsize,
  block_ysize,
  gt = c(0, 1, 0, 0, 0, 1),
  max_pixels = 0
)
```

Arguments

raster_xsize	Integer value giving the number of raster columns.
raster_ysize	Integer value giving the number of raster rows.
block_xsize	Integer value giving the horizontal size of a raster block in number of pixels.
block_ysize	Integer value giving the vertical size of a raster block in number of pixels.
gt	A numeric vector of length six containing the affine geotransform for the raster (defaults to c(0, 1, 0, 0, 0, 1)). Required only if geospatial bounding boxes of the chunks are needed in the output.
max_pixels	Numeric value (a whole number), optionally carrying the bit64::integer64 class attribute. Specifies the maximum number of pixels per chunk. Can be set to zero to define chunks as the blocks.

Details

The stand-alone function here supports the general case of chunking/tiling a raster layout without using a dataset object. If the max_pixels argument is set to zero, the chunks are raster blocks (the internal tiles in the case of a tiled format). Otherwise, chunks are defined as the maximum number of consecutive whole blocks containing <= max_pixels, that may span one or multiple whole rows of blocks.

Value

A numeric matrix with number of rows equal to the number of chunks, and named columns: xchunkoff, ychunkoff, xoff, yoff, xsize, ysize, xmin, xmax, ymin, ymax. Offsets are 0-based.

See Also

Methods `make_chunk_index()`, `readChunk()` and `writeChunk()` in class GDALRaster.

Usage example in the web article GDAL Block Cache.

Examples

```
## chunk as one block
blocks <- make_chunk_index(raster_xsize = 156335, raster_ysize = 101538,
                           block_xsize = 256, block_ysize = 256,
                           gt = c(-2362395, 30, 0, 3267405, 0, -30),
                           max_pixels = 0)

nrow(blocks)

head(blocks)

tail(blocks)

## chunk as 16 consectutive blocks
chunks <- make_chunk_index(raster_xsize = 156335, raster_ysize = 101538,
                           block_xsize = 256, block_ysize = 256,
                           gt = c(-2362395, 30, 0, 3267405, 0, -30),
                           max_pixels = 256 * 256 * 16)

nrow(chunks)

head(chunks)

tail(chunks)
```

---

mdim_as_classic *Return a view of an MDArray as a "classic" GDALDataset (i.e., 2D)*

Description

`mdim_as_classic()` returns a 2D raster view on an MDArray in a GDAL Multidimensional Raster dataset, as an object of class `GDALRaster`. Only 2D or more arrays are supported. In the case of > 2D arrays, additional dimensions will be represented as raster bands. Requires GDAL >= 3.2.

Usage

```
mdim_as_classic(
  dsn,
  array_name,
  idx_xdim,
  idx_ydim,
  read_only = TRUE,
  group_name = NULL,
```

```
  view_expr = NULL,
  allowed_drivers = NULL,
  open_options = NULL
)
```

Arguments

dsn	Character string giving the data source name of the multidimensional raster (e.g., file, VSI path).
array_name	Character string giving the name of the MDarray in dsn.
idx_xdim	Integer value giving the index of the dimension that will be used as the X/width axis (0-based).
idx_ydim	Integer value giving the index of the dimension that will be used as the Y/height axis (0-based).
read_only	Logical value, TRUE to open the dataset in read-only mode (the default).
group_name	Optional character string giving the fully qualified name of a group containing array_name.
view_expr	Optional character string giving an expression for basic array slicing and indexing, or field access (see section View Expressions).
allowed_drivers	Optional character vector of driver short names that must be considered. By default, all known multidimensional raster drivers are considered.
open_options	Optional character vector of format-specific dataset open options as "NAME=VALUE" pairs.

Value

An object of class GDALRaster.

View Expressions

A character string can be passed in argument view_expr to specify array slicing or field access. The slice expression uses the same syntax as NumPy basic slicing and indexing (0-based), or it can use field access by name. See https://numpy.org/doc/stable/user/basics.indexing.html.

GDAL support for view expression on an MDArray is documented for GDALMDArray::GetView() (see https://gdal.org/en/stable/api/gdalmdarray_cpp.html) and copied here:

Multiple [] bracket elements can be concatenated, with a slice expression or field name inside each.

For basic slicing and indexing, inside each [] bracket element, a list of indexes that apply to successive source dimensions, can be specified, using integer indexing (e.g. 1), range indexing (start:stop:step), ellipsis (...) or newaxis, using a comma separator.

Example expressions with a 2-dimensional array whose content is [[0,1,2,3],[4,5,6,7]].

- "[1][2]": returns a 0-dimensional/scalar array with the value at index 1 in the first dimension, and index 2 in the second dimension from the source array. That is, 5.
- "[1,2]": same as above, but a bit more performant.

- "[1]": returns a 1-dimensional array, sliced at index 1 in the first dimension. That is [4,5,6,7].
- "[:,2]": returns a 1-dimensional array, sliced at index 2 in the second dimension. That is [2,6].
- "[:,2:3:]": returns a 2-dimensional array, sliced at index 2 in the second dimension. That is [[2],[6]].
- "[::,2]": Same as above.
- "[...,2]": same as above, in that case, since the ellipsis only expands to one dimension here.
- "[:,::2]": returns a 2-dimensional array, with even-indexed elements of the second dimension. That is [[0,2],[4,6]].
- "[:,1::2]": returns a 2-dimensional array, with odd-indexed elements of the second dimension. That is [[1,3],[5,7]].
- "[:,1:3:]": returns a 2-dimensional array, with elements of the second dimension with index in the range [1,3]. That is [[1,2],[5,6]].
- "[::-1,:]": returns a 2-dimensional array, with the values in first dimension reversed. That is [[4,5,6,7],[0,1,2,3]].
- "[newaxis,...]": returns a 3-dimensional array, with an additional dimension of size 1 put at the beginning. That is [[[0,1,2,3],[4,5,6,7]]].

One difference with NumPy behavior is that ranges that would result in zero elements are not allowed (dimensions of size 0 not being allowed in the GDAL multidimensional model).

For field access, the syntax to use is "['field_name']". Multiple field specification is not supported currently. Both type of access can be combined, e.g. "[1]['field_name']".

Note

The indexing of array dimensions is 0-based consistent with the <ARRAY-SPEC> notation that may be used with GDAL CLI commands, e.g., gdal_usage("mdim convert") (CLI bindings require GDAL > 3.11.3). See https://gdal.org/en/stable/programs/gdal_mdim_convert.html.

Once the returned GDALRaster object has been closed, it cannot be re-opened with its $open() method.

See Also

GDALRaster-class, mdim_info(), mdim_translate()

Examples

```
f <- system.file("extdata/byte.nc", package="gdalraster")
# mdim_info(f)

(ds <- mdim_as_classic(f, "Band1", 1, 0))

plot_raster(ds, interpolate = FALSE, legend = TRUE, main = "Band1")

ds$close()
```

mdim_info *Report structure and content of a multidimensional dataset*

Description

mdim_info() is an interface to the gdalmdiminfo command-line utility (see https://gdal.org/en/stable/programs/gdalmdiminfo.html). This function lists various information about a GDAL supported multidimensional raster dataset as JSON output. It follows the JSON schema gdalmdim-info_output.schema.json. Requires GDAL >= 3.2.

Usage

```
mdim_info(
  dsn,
  array_name = "",
  pretty = TRUE,
  detailed = FALSE,
  limit = -1L,
  stats = FALSE,
  array_options = NULL,
  allowed_drivers = NULL,
  open_options = NULL,
  cout = TRUE
)
```

Arguments

dsn	Character string giving the data source name of the multidimensional raster (e.g., file, VSI path).
array_name	Character string giving the name of the MDarray in dsn.
pretty	Logical value, FALSE to output a single line without any indentation. Defaults to TRUE.
detailed	Logical value, TRUE for verbose output. Report attribute data types and array values. Defaults to FALSE.
limit	Integer value. Number of values in each dimension that is used to limit the display of array values. By default, unlimited. Only taken into account if used with detailed = TRUE. Set to a positive integer to enable.
stats	Logical value, TRUE to read and display array statistics. Forces computation if no statistics are stored in an array. Defaults to FALSE.
array_options	Optional character vector of "NAME=VALUE" pairs to filter reported arrays. Such option is format specific. Consult driver documentation (passed to GDALGroup::GetMDArrayNames()).
allowed_drivers	Optional character vector of driver short names that must be considered when opening dsn. It is generally not necessary to specify it, but it can be used to skip automatic driver detection, when it fails to select the appropriate driver.

open_options	Optional character vector of format-specific dataset openoptions as "NAME=VALUE" pairs.
cout	Logical value, TRUE to print info to the console (the default), or FALSE to suppress console output.

Value

Invisibly, a JSON string containing information about the multidimensional raster dataset. By default, the info string is also printed to the console unless cout is set to FALSE.

See Also

mdim_as_classic(), mdim_translate()

Examples

```
f <- system.file("extdata/byte.nc", package="gdalraster")
mdim_info(f)
```

---

mdim_translate *Convert multidimensional data between different formats, and subset*

Description

mdim_translate() is an interface to the gdalmdimtranslate command-line utility (see https://gdal.org/en/stable/programs/gdalmdimtranslate.html). This function converts multidimensional data between different formats and performs subsetting. Requires GDAL >= 3.2.

Usage

```
mdim_translate(
  src_dsn,
  dst_dsn,
  output_format = "",
  creation_options = NULL,
  array_specs = NULL,
  group_specs = NULL,
  subset_specs = NULL,
  scaleaxes_specs = NULL,
  allowed_drivers = NULL,
  open_options = NULL,
  strict = FALSE,
  quiet = FALSE
)
```

Arguments

src_dsn	Character string giving the name of the source multidimensional raster dataset (e.g., file, VSI path).
dst_dsn	Character string giving the name of the destination multidimensional raster dataset (e.g., file, VSI path).
output_format	Character string giving the output format (driver short name). This can be a format that supports multidimensional output (such as NetCDF: Network Common Data Form, Multidimensional VRT), or a "classic" 2D format, if only one single 2D array results from the other specified conversion operations. When this option is not specified (i.e., empty string ""), the format is guessed when possible from the extension of dst_dsn.
creation_options	Optional character vector of format-specific creation options as "NAME=VALUE" pairs. A list of options supported for a format can be obtained with getCreationOptions(), but the documentation for the format is the definitive source of information on driver creation options (see https://gdal.org/en/stable/drivers/raster/index.html). Array-level creation options may be passed by prefixing them with ARRAY: (see Details).
array_specs	Optional character vector of one or more array specifications, instead of converting the whole dataset (see Details).
group_specs	Optional character vector of one or more array specifications, instead of converting the whole dataset (see Details).
subset_specs	Optional character vector of one or more subset specifications, that perform trimming or slicing along a dimension, provided that it is indexed by a 1D variable of numeric or string data type, and whose values are monotonically sorted (see Details).
scaleaxes_specs	Optional character string for a scale-axes specification, that apply an integral scale factor to one or several dimensions, i.e., extract 1 value every N values (without resampling) (see Details).
allowed_drivers	Optional character vector of driver short names that must be considered when opening src_dsn. It is generally not necessary to specify it, but it can be used to skip automatic driver detection, when it fails to select the appropriate driver.
open_options	Optional character vector of format-specific dataset open options for src_dsn as "NAME=VALUE" pairs.
strict	Logical value, FALSE (the default) some failures during the translation are tolerated, such as not being able to write group attributes. If set to TRUE, such failures will cause the process to fail.
quiet	Logical value, set to TRUE to disable progress reporting. Defaults to FALSE.

Details

Array creation options: Array creation options must be prefixed with ARRAY:. The scope may be further restricted to arrays of a certain dimension by adding IF(DIM={ndims}): after ARRAY:.

For example, "ARRAY:IF(DIM=2):BLOCKSIZE=256,256" will restrict BLOCKSIZE=256,256 to arrays of dimension 2. Restriction to arrays of a given name is done with adding IF(NAME={name}): after ARRAY:. {name} can also be a fully qualified name. A non-driver specific array option, "AUTOSCALE=YES" can be used to ask (non indexing) variables of type Float32 or Float64 to be scaled to UInt16 with scale and offset values being computed from the minimum and maximum of the source array. The integer data type used can be set with "AUTOSCALE_DATA_TYPE=Byte|UInt16|Int16|UInt32|Int32

array_specs: Instead of converting the whole dataset, select one or more arrays, and possibly perform operations on them. One or more array specifications can be given as elements of a character vector.

An array specification may be just an array name, potentially using a fully qualified syntax ("/group/subgroup/array_name"). Or it can be a combination of options with the syntax:

"name={src_array_name}[,dstname={dst_array_name}][,resample=yes][,transpose=[{axis1},{axis2},...][

The following options are processed in that order:

- resample=yes asks for the array to run through GDALMDArray::GetResampled().
- [{axis1},{axis2},...] is the argument of GDALMDArray::Transpose(). For example, transpose=[1,0] switches the axis order of a 2D array.
- {view_expr} is the value of the viewExpr argument of GDALMDArray::GetView(). When specifying a view_expr that performs a slicing or subsetting on a dimension, the equivalent operation will be applied to the corresponding indexing variable. See ?mdim_as_classic for details on view expressions.

group_specs: Instead of converting the whole dataset, select one or more groups, and possibly perform operations on them. One or more group specifications can be given in a character vector, to operate on different groups. If only one group is specified, its content will be copied directly to the target root group. If several are specified, they are copied under the target root group.

A group specification may be just a group name, potentially using a fully qualified syntax ("/group/subgroup/subsubgrou
Or it can be a combination of options with the syntax:

"name={src_group_name}[,dstname={dst_group_name}][,recursive=no]"

subset_specs: Perform subsetting (trimming or slicing) operations along dimensions, provided that the dimension is indexed by a 1D variable of numeric or string data type, and whose values are monotonically sorted. One or more subset specifications can be given in a character vector. A subset specification string follows exactly the OGC WCS 2.0 KVP encoding for subsetting.

Syntax is "dim_name(min_val,max_val)" or "dim_name(sliced_val)". The first syntax will subset the dimension dim_name to values in the [min_val,max_val] range. The second syntax will slice the dimension dim_name to value sliced_val (and this dimension will be removed from the arrays that reference to it)

Using a subset specification is incompatible with specifying a view option in array_specs.

scaleaxes_specs: Applies an integral scale factor to one or several dimensions, i.e., extract 1 value every N values (without resampling). A scale-axes specification string follows exactly the syntax of the KVP encoding of the SCALEAXES parameter of OGC WCS 2.0 Scaling Extension, but limited to integer scale factors.

Syntax is a character string of the form:

"<dim1_name>(<scale_factor>)[,<dim2_name>(<scale_factor>)]..."

Using a scale-axes specification is incompatible with specifying a view option in array_specs.

Value

Logical value indicating success (invisible TRUE, output written to dst_dsn). An error is raised if the operation fails.

See Also

mdim_as_classic(), mdim_info()

Examples

```
f_src <- system.file("extdata/byte.nc", package="gdalraster")

## COMPRESS option for MDArray creation
opt <- NULL
if (isTRUE(gdal_get_driver_md("netCDF")$NETCDF_HAS_NC4 == "YES")) {
  # 4 x 4 is for illustration only, the sample dataset is only 20 x 20
  opt <- c("ARRAY:IF(DIM=2):BLOCKSIZE=4,4",
           "ARRAY:IF(NAME=Band1):COMPRESS=DEFLATE",
           "ARRAY:ZLEVEL=6")
}

f_dst <- tempfile(fileext = ".nc")
mdim_translate(f_src, f_dst, creation_options = opt)
info <- mdim_info(f_dst, cout = FALSE) |> yyjsonr::read_json_str()
# str(info)
info$structural_info
info$arrays$Band1$block_size
info$arrays$Band1$structural_info

(ds <- mdim_as_classic(f_dst, "Band1", 1, 0))

plot_raster(ds, interpolate = FALSE, legend = TRUE, main = "Band1")

ds$close()

## slice along the Y axis with array view
f_dst2 <- tempfile(fileext = ".nc")
mdim_translate(f_src, f_dst2, array_specs = "name=Band1,view=[10:20,...]")
(ds <- mdim_as_classic(f_dst2, "Band1", 1, 0))

plot_raster(ds, interpolate = FALSE, legend = TRUE,
            main = "Band1[10:20,...]")

ds$close()

## trim X and Y by subsetting
f_dst3 <- tempfile(fileext = ".nc")
subsets <- c("x(441000,441800)", "y(3750400,3751000)")
mdim_translate(f_src, f_dst3, subset_specs = subsets)
(ds <- mdim_as_classic(f_dst3, "Band1", 1, 0))

plot_raster(ds, interpolate = FALSE, legend = TRUE,
```

```
            main = "Band1 trimmed")

ds$close()

## subsample along X and Y
f_dst4 <- tempfile(fileext = ".nc")
mdim_translate(f_src, f_dst4, scaleaxes_specs = "x(2),y(2)")
(ds <- mdim_as_classic(f_dst4, "Band1", 1, 0))

plot_raster(ds, interpolate = FALSE, legend = TRUE,
            main = "Band1 subsampled")

ds$close()
```

---

ogr2ogr *Convert vector data between different formats*

Description

ogr2ogr() is a wrapper of the ogr2ogr command-line utility (see https://gdal.org/en/stable/programs/ogr2ogr.html). This function can be used to convert simple features data between file formats. It can also perform various operations during the process, such as spatial or attribute selection, reducing the set of attributes, setting the output coordinate system or even reprojecting the features during translation. Refer to the GDAL documentation at the URL above for a description of command-line arguments that can be passed in cl_arg.

Usage

```
ogr2ogr(
  src_dsn,
  dst_dsn,
  src_layers = NULL,
  cl_arg = NULL,
  open_options = NULL
)
```

Arguments

src_dsn	Character string. Data source name of the source vector dataset.
dst_dsn	Character string. Data source name of the destination vector dataset.
src_layers	Optional character vector of layer names in the source dataset. Defaults to all layers.

cl_arg	Optional character vector of command-line arguments for the GDAL ogr2ogr command-line utility (see URL above).
open_options	Optional character vector of dataset open options.

Value

Logical indicating success (invisible TRUE). An error is raised if the operation fails.

Note

For progress reporting, see command-line argument -progress: Display progress on terminal. Only works if input layers have the "fast feature count" capability.

See Also

ogrinfo(), the ogr_manage utilities

translate() for raster data

Examples

```
src <- system.file("extdata/ynp_fires_1984_2022.gpkg", package="gdalraster")

# Convert GeoPackage to Shapefile
ynp_shp <- file.path(tempdir(), "ynp_fires.shp")
ogr2ogr(src, ynp_shp, src_layers = "mtbs_perims")

# Reproject to WGS84
ynp_gpkg <- file.path(tempdir(), "ynp_fires.gpkg")
args <- c("-t_srs", "EPSG:4326", "-nln", "fires_wgs84")
ogr2ogr(src, ynp_gpkg, cl_arg = args)

# Clip to a bounding box (xmin, ymin, xmax, ymax in the source SRS)
# This will select features whose geometry intersects the bounding box.
# The geometries themselves will not be clipped unless "-clipsrc" is
# specified.
# The source SRS can be overridden with "-spat_srs" "<srs_def>"
# Using -update mode to write a new layer in the existing DSN
bb <- c(469685.97, 11442.45, 544069.63, 85508.15)
args <- c("-update", "-nln", "fires_clip", "-spat", bb)
ogr2ogr(src, ynp_gpkg, cl_arg = args)

# Filter features by a -where clause
sql <- "ig_year >= 2000 ORDER BY ig_year"
args <- c("-update", "-nln", "fires_2000-2020", "-where", sql)
ogr2ogr(src, ynp_gpkg, src_layers = "mtbs_perims", cl_arg = args)

# Dissolve features based on a shared attribute value
if (has_spatialite()) {
    sql <- "SELECT ig_year, ST_Union(geom) AS geom
              FROM mtbs_perims GROUP BY ig_year"
    args <- c("-update", "-sql", sql, "-dialect", "SQLITE")
```

```
    args <- c(args, "-nlt", "MULTIPOLYGON", "-nln", "dissolved_on_year")
    ogr2ogr(src, ynp_gpkg, cl_arg = args)
}
```

---

ogrinfo *Retrieve information about a vector data source*

Description

ogrinfo() is a wrapper of the ogrinfo command-line utility (see https://gdal.org/en/stable/programs/ogrinfo.html). This function lists information about an OGR-supported data source. It is also possible to edit data with SQL statements. Refer to the GDAL documentation at the URL above for a description of command-line arguments that can be passed in cl_arg. Requires GDAL >= 3.7.

Usage

```
ogrinfo(
  dsn,
  layers = NULL,
  cl_arg = as.character(c("-so", "-nomd")),
  open_options = NULL,
  read_only = TRUE,
  cout = TRUE
)
```

Arguments

dsn	Character string. Data source name (e.g., filename, database connection string, etc.)
layers	Optional character vector of layer names in the source dataset.
cl_arg	Optional character vector of command-line arguments for the ogrinfo command-line utility in GDAL (see URL above for reference). The default is c("-so", "-nomd") (see Note).
open_options	Optional character vector of dataset open options.
read_only	Logical scalar. TRUE to open the data source read-only (the default), or FALSE to open with write access.
cout	Logical scalar. TRUE to write info to the standard C output stream (the default). FALSE to suppress console output.

Value

Invisibly, a character string containing information about the vector dataset, or empty string ("") in case of error.

Note

The command-line argument `-so` provides a summary only, i.e., does not include details about every single feature of a layer. `-nomd` suppresses metadata printing. Some datasets may contain a lot of metadata strings.

See Also

`ogr2ogr()`, ogr_manage

Examples

```
src <- system.file("extdata/ynp_fires_1984_2022.gpkg", package="gdalraster")

# Get the names of the layers in a GeoPackage file
ogrinfo(src)

# Summary of a layer
ogrinfo(src, "mtbs_perims")

# Query an attribute to restrict the output of the features in a layer
args <- c("-ro", "-nomd", "-where", "ig_year = 2020")
ogrinfo(src, "mtbs_perims", args)

# Copy to a temporary in-memory file that is writeable
src_mem <- paste0("/vsimem/", basename(src))
vsi_copy_file(src, src_mem)

# Add a column to a layer
args <- c("-sql", "ALTER TABLE mtbs_perims ADD burn_bnd_ha float")
ogrinfo(src_mem, cl_arg = args, read_only = FALSE)

# Update values of the column with SQL and specify a dialect
sql <- "UPDATE mtbs_perims SET burn_bnd_ha = (burn_bnd_ac / 2.471)"
args <- c("-dialect", "sqlite", "-sql", sql)
ogrinfo(src_mem, cl_arg = args, read_only = FALSE)
```

---

ogr_define *OGR feature class definition for vector data*

Description

This topic contains documentation and helper functions for defining an OGR feature class. A named list containing zero or more attribute field definitions, along with one or more geometry field definitions, comprise an OGR feature class definition (a.k.a. layer definition). `ogr_def_layer()` initializes such a list with the geometry type and (optionally) a spatial reference system. Attribute fields may then be added to the layer definition. `ogr_def_field()` creates an attribute field definition, a list containing the field's data type and potentially other optional field properties.

ogr_def_geom_field() similarly creates a geometry field definition. This might be used with certain vector formats that support multiple geometry columns (e.g., PostGIS). ogr_def_field_domain() creates a field domain definition. A field domain is a set of constraints that apply to one or several fields. This is a concept found, e.g., in ESRI File Geodatabase and in GeoPackage via the Schema extension (see https://github.com/OSGeo/gdal/pull/3638). GDAL >= 3.3 supports reading and writing field domains with certain drivers (e.g., GPKG and OpenFileGDB).

Usage

```
ogr_def_layer(geom_type, geom_fld_name = "geom", srs = NULL)

ogr_def_field(
  fld_type,
  fld_subtype = NULL,
  fld_width = NULL,
  fld_precision = NULL,
  is_nullable = NULL,
  is_unique = NULL,
  default_value = NULL,
  domain_name = NULL
)

ogr_def_geom_field(geom_type, srs = NULL, is_nullable = NULL)

ogr_def_field_domain(
  domain_type,
  domain_name,
  description = NULL,
  fld_type = NULL,
  fld_subtype = "OFSTNone",
  split_policy = "DEFAULT_VALUE",
  merge_policy = "DEFAULT_VALUE",
  coded_values = NULL,
  range_min = NULL,
  min_is_inclusive = TRUE,
  range_max = NULL,
  max_is_inclusive = TRUE,
  glob = ""
)
```

Arguments

geom_type	Character string specifying a geometry type (see Details).
geom_fld_name	Character string specifying a geometry field name Defaults to "geom".
srs	Character string containing a spatial reference system definition as OGC WKT or other well-known format (e.g., the input formats usable with srs_to_wkt()).
fld_type	Character string containing the name of a field data type (e.g., "OFTInteger", "OFTInteger64", "OFTReal", "OFTString", ...).

fld_subtype	Character string containing the name of a field subtype. One of "OFSTNone" (the default), "OFSTBoolean", "OFSTInt16", "OFSTFloat32", "OFSTJSON", "OFSTUUID".
fld_width	Optional integer value specifying max number of characters.
fld_precision	Optional integer value specifying number of digits after the decimal point.
is_nullable	Optional NOT NULL field constraint (logical value). Defaults to TRUE.
is_unique	Optional UNIQUE constraint on the field (logical value). Defaults to FALSE.
default_value	Optional default value for the field given as a character string.
domain_name	Character string specifying the name of the field domain. Optional for ogr_def_field(), required for ogr_def_field_domain().
domain_type	Character string specifying a field domain type (see Details). Must be one of "Coded", "Range", "RangeDateTime", "GLOB" (case-insensitive).
description	Optional character string giving a description of the field domain.
split_policy	Character string specifying the split policy of the field domain. One of "DEFAULT_VALUE", "DUPLICATE", "GEOMETRY_RATIO" (supported by ESRI File Geodatabase format via OpenFileGDB driver).
merge_policy	Character string specifying the merge policy of the field domain. One of "DEFAULT_VALUE", "SUM", "GEOMETRY_WEIGHTED" (supported by ESRI File Geodatabase format via OpenFileGDB driver).
coded_values	Either a vector of the allowed codes, or character vector of "CODE=VALUE" pairs (the expanded "value" associated with a code is optional), or a two-column data frame with (codes, values). If data frame, the second column of values must be character type (i.e., the descriptive text for each code). This argument is required if domain_type = "Coded". Each code should appear only once, but it is the responsibility of the user to check it.
range_min	Minimum value in a Range or RangeDateTime field domain (can be NULL). The data type must be consistent with with the field type given in the fld_type argument.
min_is_inclusive	= Logical value, whether the minimum value is included in the range. Defaults to TRUE. Required argument if domain_type is "Range" or "RangeDateTime".
range_max	Maximum value in a Range or RangeDateTime field domain (can be NULL). The data type must be consistent with with the field type given in the fld_type argument.
max_is_inclusive	= Logical value, whether the maximum value is included in the range. Defaults to TRUE. Required argument if domain_type is "Range" or "RangeDateTime".
glob	Character string containing the GLOB expression. Required if domain_type is "GLOB".

Details

All features in an OGR Layer share a common schema (feature class), modeled in GDAL by its OGRFeatureDefn class. A feature class definition includes the set of attribute fields and their data types and the geometry field(s). In R, a feature class definition is represented as a named list,

with names being the attribute/geometry field names, and each list element holding an attribute or geometry field definition.

The definition for an attribute field is a named list with elements:

```
$type       : OGR Field Type ("OFTReal", "OFTString" etc.)
$subtype    : optional ("OFSTBoolean", ...)
$width      : optional max number of characters
$precision  : optional number of digits after the decimal point
$is_nullable: optional NOT NULL constraint (logical value)
$is_unique  : optional UNIQUE constraint (logical value)
$default    : optional default value as character string
$domain     : optional field domain name
$is_geom    : FALSE (the default) for attribute fields
```

An OGR field type is specified as a character string with possible values: `OFTInteger`, `OFTIntegerList`, `OFTReal`, `OFTRealList`, `OFTString`, `OFTStringList`, `OFTBinary`, `OFTDate`, `OFTTime`, `OFTDateTime`, `OFTInteger64`, `OFTInteger64List`.

An optional field subtype is specified as a character string with possible values: `OFSTNone`, `OFSTBoolean`, `OFSTInt16`, `OFSTFloat32`, `OFSTJSON`, `OFSTUUID`.

By default, fields are nullable and have no unique constraint. Not-null and unique constraints are not supported by all format drivers.

A default field value is taken into account by format drivers (generally those with a SQL interface) that support it at field creation time. If given in the field definition, `default` must be a character string. The accepted values are `"NULL"`, a numeric value (e.g., `"0"`), a literal value enclosed between single quote characters (e.g., `"'a default value'"`, with any inner single quote characters escaped by repetition of the single quote character), `"CURRENT_TIMESTAMP"`, `"CURRENT_TIME"`, `"CURRENT_DATE"` or a driver-specific expression (that might be ignored by other drivers). For a datetime literal value, format should be `"'YYYY/MM/DD HH:MM:SS[.sss]'"` (considered as UTC time).

The definition for a geometry field is a named list with elements:

```
$type       : geom type ("Point", "Polygon", etc.)
$srs        : optional spatial reference as WKT string
$is_nullable: optional NOT NULL constraint (logical value)
$is_geom    : TRUE (required) for geometry fields
```

Typically, there is one geometry field on a layer, but some formats support more than one geometry column per table (e.g., "PostgreSQL / PostGIS" and "SQLite / Spatialite RDBMS").

Geometry types are specified as a character string containing OGC WKT. Common types include: `Point`, `LineString`, `Polygon`, `MultiPoint`, `MultiLineString`, `MultiPolygon`. See the GDAL documentation for a list of all supported geometry types:
https://gdal.org/en/stable/api/vector_c_api.html#_CPPv418OGRwkbGeometryType

Format drivers may or may not support not-null constraints on attribute and geometry fields. If they support creating fields with not-null constraints, this is generally before creating any features to the layer. In some cases, a not-null constraint may be available as a layer creation option. For example, GeoPackage format has a layer creation option `GEOMETRY_NULLABLE=[YES/NO]`.

The definition for a field domain is a named list with elements:

```
$type             : domain type ("Coded", "Range", "RangeDateTime", "GLOB")
$domain_name      : name of the field domain (character string)
$description      : optional domain description (character string)
$field_type       : OGR Field Type (see attribute field definitions above)
$field_subtype    : optional OGR Field Subtype ("OFSTBoolean", ...)
$split_policy     : split policy of the field domain (see below)
$merge_policy     : merge policy of the field domain (see below)
$coded_values     : vector of allowed codes, or data frame of (codes, values)
$min_value        : minimum value (data type compatible with $field_type)
$min_is_inclusive : whether the minimum value is included in the range
$max_value        : maximum value (data type compatible with $field_type)
$max_is_inclusive : whether the maximum value is included in the range
$glob             : GLOB expression (character string)
```

A field domain can be one of three types:

- `Coded`: an enumerated list of allowed codes with their descriptive values
- `Range`: a range constraint (min, max)
- `GLOB`: a GLOB expression (matching expression like `"*[a-z][0-1]?"`)

`type` can also be specified as `"RangeDateTime"`, a range constraint for `OFTDateTime` fields with `min_value` and `max_value` given as `"POSIXct"` DateTimes.

Split and merge policies are supported by ESRI File Geodatabase format via OpenFileGDB driver (or FileGDB driver dependent on FileGDB API library). When a feature is split in two, `split_policy` defines how the value of attributes following the domain are computed. Possible values are `"DEFAULT_VALUE"` (default value), `"DUPLICATE"` (duplicate), and `"GEOMETRY_RATIO"` (new values are computed by the ratio of their area/length compared to the area/length of the original feature). When a feature is built by merging two features, `merge_policy` defines how the value of attributes following the domain are computed. Possible values are `"DEFAULT_VALUE"` (default value), `"SUM"` (sum), and `"GEOMETRY_WEIGHTED"` (new values are computed as the weighted average of the source values).

Note

The feature id (FID) is a special property of a feature and not treated as an attribute of the feature. Additional information is given in the GDAL documentation for the OGR SQL and SQLite SQL dialects. Implications for SQL statements and result sets may depend on the dialect used.

Some vector formats do not support schema definition prior to creating features. For example, with GeoJSON only the *Feature* object has a member with name *properties*. The specification does not require all *Feature* objects in a collection to have the same schema of properties, nor does it require all *Feature* objects in a collection to have geometry of the same type (https://geojson.org/).

See Also

`ogr_ds_create()`, `ogr_layer_create()`, `ogr_field_create()`

WKT representation of geometry:
https://en.wikipedia.org/wiki/Well-known_text_representation_of_geometry

Field domains:
https://desktop.arcgis.com/en/arcmap/latest/manage-data/geodatabases/an-overview-of-attribute-domai

htm
https://www.geopackage.org/spec/#extension_schema
https://gdal.org/en/stable/doxygen/classOGRFieldDomain.html#details

Examples

```
# create a SQLite data source, with SpatiaLite extensions if available
dsn <- file.path(tempdir(), "test.sqlite")
opt <- NULL
if (has_spatialite()) {
  opt <- "SPATIALITE=YES"
}

# This creates an empty data source. Note that we could also create a layer
# at the same time in this function call, but for this example we do that
# separately, to show creation of a layer on an existing data source.
ogr_ds_create("SQLite", dsn, dsco = opt)

# define a layer
defn <- ogr_def_layer("Point", srs = "EPSG:4326")
defn$my_id <- ogr_def_field("OFTInteger64")
defn$my_description <- ogr_def_field("OFTString")

# create a layer in the existing data source
ogr_ds_test_cap(dsn)$CreateLayer  # TRUE

ogr_layer_create(dsn, "layer1", layer_defn = defn)

ogr_ds_layer_names(dsn)

ogr_layer_field_names(dsn, "layer1")

deleteDataset(dsn)
```

---

ogr_manage *Utility functions for managing vector data sources*

Description

This set of functions can be used to create new vector datasets, test existence of dataset/layer/field, test dataset and layer capabilities, create new layers in an existing dataset, get the names of field domains stored in a dataset, write field domains in a dataset, delete layers, create new attribute and geometry fields on an existing layer, rename and delete fields, and edit data with SQL statements.

Usage

```
ogr_ds_exists(dsn, with_update = FALSE)

ogr_ds_format(dsn)
```

```
ogr_ds_test_cap(dsn, with_update = TRUE)

ogr_ds_create(
  format,
  dsn,
  layer = NULL,
  layer_defn = NULL,
  geom_type = NULL,
  srs = NULL,
  fld_name = NULL,
  fld_type = NULL,
  dsco = NULL,
  lco = NULL,
  overwrite = FALSE,
  return_obj = FALSE
)

ogr_ds_layer_count(dsn)

ogr_ds_layer_names(dsn)

ogr_ds_field_domain_names(dsn)

ogr_ds_add_field_domain(dsn, fld_dom_defn)

ogr_ds_delete_field_domain(dsn, domain_name)

ogr_layer_exists(dsn, layer)

ogr_layer_test_cap(dsn, layer = NULL, with_update = TRUE)

ogr_layer_create(
  dsn,
  layer,
  layer_defn = NULL,
  geom_type = NULL,
  srs = NULL,
  lco = NULL,
  return_obj = FALSE
)

ogr_layer_field_names(dsn, layer = NULL)

ogr_layer_rename(dsn, layer, new_name)

ogr_layer_delete(dsn, layer)
```

```
ogr_field_index(dsn, layer, fld_name)

ogr_field_create(
  dsn,
  layer,
  fld_name,
  fld_defn = NULL,
  fld_type = "OFTInteger",
  fld_subtype = "OFSTNone",
  fld_width = 0L,
  fld_precision = 0L,
  is_nullable = TRUE,
  is_unique = FALSE,
  default_value = "",
  domain_name = NULL
)

ogr_geom_field_create(
  dsn,
  layer,
  fld_name,
  geom_fld_defn = NULL,
  geom_type = NULL,
  srs = NULL,
  is_nullable = TRUE
)

ogr_field_rename(dsn, layer, fld_name, new_name)

ogr_field_set_domain_name(dsn, layer, fld_name, domain_name)

ogr_field_delete(dsn, layer, fld_name)

ogr_execute_sql(dsn, sql, spatial_filter = NULL, dialect = NULL)
```

Arguments

dsn	Character string. The vector data source name, e.g., a filename or database connection string.
with_update	Logical value. TRUE to request update access when opening the dataset, or FALSE to open read-only.
format	GDAL short name of the vector format as character string. Examples of some common output formats include: "GPKG", "FlatGeobuf", "ESRI Shapefile", "SQLite".
layer	Character string for a layer name in a vector dataset. The layer argument may be given as empty string ("") in which case the first layer by index will be opened (except with ogr_layer_delete() and ogr_layer_rename() for which a layer name must be specified).

layer_defn	A feature class definition for layer as a list of zero or more attribute field definitions, and at least one geometry field definition (see ogr_define). Each field definition is a list with named elements containing values for the field type element and other properties. If layer_defn is given, it will be used and any additional parameters passed that relate to the feature class definition will be ignored (i.e., geom_type and srs, as well as fld_name and fld_type in ogr_ds_create()). The first geometry field definition in layer_defn defines the geometry type and spatial reference system for the layer (the geom field definition must contain type, and should also contain srs when creating a layer from a feature class definition).
geom_type	Character string specifying a geometry type (see Details).
srs	Character string containing a spatial reference system definition as OGC WKT or other well-known format (e.g., the input formats usable with srs_to_wkt()).
fld_name	Character string containing the name of an attribute field in layer.
fld_type	Character string containing the name of a field data type (e.g., "OFTInteger", "OFTInteger64", "OFTReal", "OFTString", ...).
dsco	Optional character vector of format-specific creation options for dsn ("NAME=VALUE" pairs).
lco	Optional character vector of format-specific creation options for layer ("NAME=VALUE" pairs).
overwrite	Logical value. TRUE to overwrite dsn if it already exists when calling ogr_ds_create(). Default is FALSE.
return_obj	Logical value. If TRUE, an object of class GDALVector open on the newly created layer will be returned. Defaults to FALSE. Must be used with either the layer or layer_defn arguments.
fld_dom_defn	A field domain definition, i.e., a list generated with ogr_def_field_domain().
domain_name	Character string specifying the name of the field domain.
new_name	Character string containing a new name to assign.
fld_defn	A field definition as list (see ogr_def_field()). Additional arguments in ogr_field_create() will be ignored if a fld_defn is given.
fld_subtype	Character string containing the name of a field subtype. One of "OFSTNone" (the default), "OFSTBoolean", "OFSTInt16", "OFSTFloat32", "OFSTJSON", "OFSTUUID".
fld_width	Optional integer value specifying max number of characters.
fld_precision	Optional integer value specifying number of digits after the decimal point.
is_nullable	Optional NOT NULL field constraint (logical value). Defaults to TRUE.
is_unique	Optional UNIQUE constraint on the field (logical value). Defaults to FALSE.
default_value	Optional default value for the field as a character string.
geom_fld_defn	A geometry field definition as list (see ogr_def_geom_field()). Additional arguments in ogr_geom_field_create() will be ignored if a geom_fld_defn is given.
sql	Character string containing an SQL statement (see Note).

spatial_filter Either a numeric vector of length four containing a bounding box (xmin, ymin, xmax, ymax), or a character string containing a geometry as OGC WKT, representing a spatial filter.

dialect Character string specifying the SQL dialect to use. The OGR SQL engine ("OGRSQL") will be used by default if a value is not given. The "SQLite" dialect can also be used (see Note).

Details

These functions are complementary to ogrinfo() and ogr2ogr() for vector data management. Bindings to OGR wrap portions of the GDAL Vector API (ogr_core.h and ogr_api.h, https://gdal.org/en/stable/api/vector_c_api.html).

ogr_ds_exists() tests whether a vector dataset can be opened from the given data source name (DSN), potentially testing for update access. Returns a logical value.

ogr_ds_format() returns a character string containing the short name of the format driver for a given DSN, or NULL if the dataset cannot be opened as a vector source.

ogr_ds_test_cap() tests the capabilities of a vector data source, attempting to open it with update access by default. Returns a list of capabilities with values TRUE or FALSE, or NULL is returned if dsn cannot be opened with the requested access. Wrapper of GDALDatasetTestCapability() in the GDAL API. The returned list contains the following named elements:

- CreateLayer: TRUE if this dataset can create new layers
- DeleteLayer: TRUE if this dataset can delete existing layers
- CreateGeomFieldAfterCreateLayer: TRUE if the layers of this dataset supports geometry field creation just after layer creation
- CurveGeometries: TRUE if this dataset supports curve geometries
- Transactions: TRUE if this dataset supports (efficient) transactions
- EmulatedTransactions: TRUE if this dataset supports transactions through emulation
- RandomLayerRead: TRUE if this dataset has a dedicated GetNextFeature() implementation, potentially returning features from layers in a non-sequential way
- RandomLayerWrite: TRUE if this dataset supports calling CreateFeature() on layers in a non-sequential way
- AddFieldDomain: TRUE if this dataset supports adding field domains (GDAL >= 3.3)
- DeleteFieldDomain: TRUE if this dataset supports deleting field domains (GDAL >= 3.5) UpdateFieldDomain: TRUE if this dataset supports updating a an existing field domain by replacing its definition (GDAL >= 3.5)

ogr_ds_create() creates a new vector dataset, optionally also creating a layer, and optionally creating one or more fields on the layer. The attribute fields and geometry field(s) to create can be specified as a feature class definition (layer_defn as list, see ogr_define), or alternatively, by giving the geom_type and srs, optionally along with one fld_name and fld_type to be created in the layer. By default, returns logical TRUE indicating success (output written to dst_filename), or an object of class GDALVector for the output layer will be returned if return_obj = TRUE. An error is raised if the operation fails.

ogr_ds_layer_count() returns the number of layers in a vector dataset.

ogr_ds_layer_names() returns a character vector of layer names in a vector dataset, or NULL if no layers are found.

ogr_ds_field_domain_names() returns a character vector with the names of all field domains stored in the dataset. Returns NULL and emits a warning if an error occurs or if the format does not support reading field domains. Returns a character vector of length 0 (character(0)) if the format supports field domains but none are present in the dataset. Requires GDAL >= 3.5. See ogr_define for information on field domains.

ogr_ds_add_field_domain() adds a field domain to a vector dataset. Only a few drivers will support this operation, and some of them might only support it only for some types of field domains. GeoPackage and OpenFileGDB drivers support this operation. A dataset having at least some support should report the AddFieldDomain dataset capability (see ogr_ds_test_cap()above). Returns a logical value, TRUE indicating success. Requires GDAL >= 3.3.

ogr_ds_delete_field_domain() deletes a field domain from a vector dataset if supported (see ogr_ds_test_cap()above for the DeleteFieldDomain dataset capability). Returns a logical value, TRUE indicating success. Requires GDAL >= 3.5.

ogr_layer_exists() tests whether a layer can be accessed by name in a given vector dataset. Returns a logical value

ogr_layer_test_cap() tests whether a layer supports named capabilities, attempting to open the dataset with update access by default. Returns a list of capabilities with values TRUE or FALSE. NULL is returned if dsn cannot be opened with the requested access, or layer cannot be found. The returned list contains the following named elements: RandomRead, SequentialWrite, RandomWrite, UpsertFeature, FastSpatialFilter, FastFeatureCount, FastGetExtent, FastSetNextByIndex, CreateField, CreateGeomField, DeleteField, ReorderFields, AlterFieldDefn, AlterGeomFieldDefn, IgnoreFields, DeleteFeature, Rename, StringsAsUTF8, CurveGeometries. See the GDAL documentation for OGR_L_TestCapability().

ogr_layer_create() creates a new layer in an existing vector data source, with a specified geometry type and spatial reference definition. This function also accepts a feature class definition given as a list of field names and their definitions (see ogr_define). (Note: use ogr_ds_create() to create single-layer formats such as "ESRI Shapefile", "FlatGeobuf", "GeoJSON", etc.) By default, returns logical TRUE indicating success, or an object of class GDALVector will be returned if return_obj = TRUE. An error is raised if the operation fails.

ogr_layer_field_names() returns a character vector of field names on a layer, or NULL if no fields are found. The first layer by index is opened if NULL is given for the layer argument.

ogr_layer_rename() renames a layer in a vector dataset. This operation is implemented only by layers that expose the Rename capability (see ogr_layer_test_cap() above). This operation will fail if a layer with the new name already exists. Returns a logical value, TRUE indicating success. Requires GDAL >= 3.5.

ogr_layer_delete() deletes an existing layer in a vector dataset. Returns a logical value, TRUE indicating success.

ogr_field_index() tests for existence of an attribute field by name. Returns the field index on the layer (0-based), or -1 if the field does not exist.

ogr_field_create() creates a new attribute field of specified data type in a given DSN/layer. Several optional field properties can be specified in addition to the type. Returns a logical value, TRUE indicating success.

`ogr_geom_field_create()` creates a new geometry field of specified type in a given DSN/layer. Returns a logical value, `TRUE` indicating success.

`ogr_field_rename()` renames an existing field on a vector layer. Not all format drivers support this function. Some drivers may only support renaming a field while there are still no features in the layer. `AlterFieldDefn` is the relevant layer capability to check. Returns a logical value, `TRUE` indicating success.

`ogr_field_set_domain_name()` sets the field domain name for an existing attribute field on a vector layer. `AlterFieldDefn` layer capability is required. Returns a logical value, `TRUE` indicating success. Requires GDAL >= 3.3.

`ogr_field_delete()` deletes an existing field on a vector layer. Not all format drivers support this function. Some drivers may only support deleting a field while there are still no features in the layer. Returns a logical value, `TRUE` indicating success.

`ogr_execute_sql()` executes an SQL statement against the data store. This function can be used to modify the schema or edit data using SQL (e.g., `ALTER TABLE`, `DROP TABLE`, `CREATE INDEX`, `DROP INDEX`, `INSERT`, `UPDATE`, `DELETE`), or to execute a query (i.e., `SELECT`). Returns `NULL` (invisibly) for statements that are in error, or that have no results set, or an object of class `GDALVector` representing a results set from the query. Wrapper of `GDALDatasetExecuteSQL()` in the GDAL API.

Note

The OGR SQL document linked under **See Also** contains information on the SQL dialect supported internally by GDAL/OGR. Some format drivers (e.g., PostGIS) pass the SQL directly through to the underlying RDBMS (unless `OGRSQL` is explicitly passed as the dialect). The SQLite dialect can also be requested with the `SQLite` string passed as the `dialect` argument of `ogr_execute_sql()`. This assumes that GDAL/OGR is built with support for SQLite, and preferably also with Spatialite support to benefit from spatial functions. The GDAL document for SQLite dialect has detailed information.

Other SQL dialects may also be present for some vector formats. For example, the `"INDIRECT_SQLITE"` dialect might potentially be used with GeoPackage format (https://gdal.org/en/stable/drivers/vector/gpkg.html#sql).

The function `ogrinfo()` can also be used to edit data with SQL statements (GDAL >= 3.7).

The name of the geometry column of a layer is empty (`""`) with some formats such as ESRI Shapefile and FlatGeobuf. Implications for SQL may depend on the dialect used. See the GDAL documentation for the "OGR SQL" and "SQLite" dialects for details.

See Also

OGR SQL dialect and SQLite SQL dialect:
https://gdal.org/en/stable/user/ogr_sql_sqlite_dialect.html

Examples

```
## Create GeoPackage and manage schema
dsn <- file.path(tempdir(), "test1.gpkg")
ogr_ds_create("GPKG", dsn)
ogr_ds_exists(dsn, with_update = TRUE)
```

```
# dataset capabilities
ogr_ds_test_cap(dsn)

ogr_layer_create(dsn, layer = "layer1", geom_type = "Polygon",
                 srs = "EPSG:5070")

ogr_field_create(dsn, layer = "layer1",
                 fld_name = "field1",
                 fld_type = "OFTInteger64",
                 is_nullable = FALSE)
ogr_field_create(dsn, layer = "layer1",
                 fld_name = "field2",
                 fld_type = "OFTString")

ogr_ds_layer_count(dsn)
ogr_ds_layer_names(dsn)
ogr_layer_field_names(dsn, layer = "layer1")

# delete a field
if (ogr_layer_test_cap(dsn, "layer1")$DeleteField) {
  ogr_field_delete(dsn, layer = "layer1", fld_name = "field2")
}

ogr_layer_field_names(dsn, "layer1")

# define a feature class and create layer
defn <- ogr_def_layer("Point", srs = epsg_to_wkt(4326))
# add the attribute fields
defn$id_field <- ogr_def_field(fld_type = "OFTInteger64",
                               is_nullable = FALSE,
                               is_unique = TRUE)
defn$str_field <- ogr_def_field(fld_type = "OFTString",
                                fld_width = 25,
                                is_nullable = FALSE,
                                default_value = "'a default string'")
defn$numeric_field <- ogr_def_field(fld_type = "OFTReal",
                                    default_value = "0.0")

ogr_layer_create(dsn, layer = "layer2", layer_defn = defn)
ogr_ds_layer_names(dsn)
ogr_layer_field_names(dsn, layer = "layer2")

# add a field using SQL instead
ogr_execute_sql(dsn, sql = "ALTER TABLE layer2 ADD field4 float")

# rename a field
if (ogr_layer_test_cap(dsn, "layer1")$AlterFieldDefn) {
  ogr_field_rename(dsn, layer = "layer2",
                   fld_name = "field4",
                   new_name = "renamed_field")
}
ogr_layer_field_names(dsn, layer = "layer2")
```

```
# GDAL >= 3.7
if (gdal_version_num() >= gdal_compute_version(3, 7, 0))
  ogrinfo(dsn, "layer2")


## Edit data using SQL
src <- system.file("extdata/ynp_fires_1984_2022.gpkg", package="gdalraster")
perims_shp <- file.path(tempdir(), "mtbs_perims.shp")
ogr2ogr(src_dsn = src, dst_dsn = perims_shp, src_layers = "mtbs_perims")
ogr_ds_format(dsn = perims_shp)
ogr_ds_layer_names(dsn = perims_shp)
ogr_layer_field_names(dsn = perims_shp, layer = "mtbs_perims")

alt_tbl <- "ALTER TABLE mtbs_perims ADD burn_bnd_ha float"
ogr_execute_sql(dsn = perims_shp, sql = alt_tbl)

upd <- "UPDATE mtbs_perims SET burn_bnd_ha = (burn_bnd_ac / 2.471)"
ogr_execute_sql(dsn = perims_shp, sql = upd, dialect = "SQLite")
ogr_layer_field_names(dsn = perims_shp, layer = "mtbs_perims")

# if GDAL >= 3.7:
#   ogrinfo(dsn = perims_shp, layer = "mtbs_perims")
# or, for output incl. the feature data (omit the default "-so" arg):
#   ogrinfo(dsn = perims_shp, layer = "mtbs_perims", cl_arg = "-nomd")
```

---

ogr_proc *GDAL OGR facilities for vector geoprocessing*

Description

ogr_proc() performs GIS overlay operations on vector layers (https://en.wikipedia.org/wiki/Vector_overlay). It provides an interface to the GDAL API methods for these operations (OGRLayer::Intersection(), OGRLayer::Union(), etc). Inputs are given as objects of class GDALVector, which may have spatial and/or attribute filters applied. The output layer will be created if it does not exist, but output can also be appended to an existing layer, or written to an existing empty layer that has a custom schema defined. ogr_proc() is basically a port of the ogr_layer_algebra utility in the GDAL Python bindings.

Usage

```
ogr_proc(
  mode,
  input_lyr,
  method_lyr,
  out_dsn,
  out_lyr_name = NULL,
  out_geom_type = NULL,
  out_fmt = NULL,
```

```
  dsco = NULL,
  lco = NULL,
  mode_opt = NULL,
  overwrite = FALSE,
  quiet = FALSE,
  return_obj = TRUE
)
```

Arguments

mode	Character string specifying the operation to perform. One of Intersection, Union, SymDifference, Identity, Update, Clip or Erase (see Details).
input_lyr	An object of class GDALVector to use as the input layer. For overlay operations, this is the first layer in the operation.
method_lyr	An object of class GDALVector to use as the method layer. This is the conditional layer supplied to an operation (e.g., Clip, Erase, Update), or the second layer in overlay operations (e.g., Union, Intersection, SymDifference).
out_dsn	The destination vector filename or database connection string to which the output layer will be written.
out_lyr_name	Layer name where the output vector will be written. May be NULL (e.g., shapefile), but typically must be specified.
out_geom_type	Character string specifying the geometry type of the output layer. One of NONE, GEOMETRY, POINT, LINESTRING, POLYGON, GEOMETRYCOLLECTION, MULTIPOINT, MULTIPOLYGON, GEOMETRY25D, POINT25D, LINESTRING25D, POLYGON25D, GEOMETRYCOLLECTION25D, MULTIPOINT25D, MULTIPOLYGON25D. Defaults to UNKNOWN if not specified.
out_fmt	GDAL short name of the output vector format. If unspecified, the function will attempt to guess the format from the value of out_dsn.
dsco	Optional character vector of format-specific creation options for out_dsn ("NAME=VALUE" pairs).
lco	Optional character vector of format-specific creation options for out_layer ("NAME=VALUE" pairs).
mode_opt	Optional character vector of "NAME=VALUE" pairs that specify processing options. Available options depend on the value of mode (see Details).
overwrite	Logical value. TRUE to overwrite the output layer if it already exists. Defaults to FALSE.
quiet	Logical value. If TRUE, a progress bar will not be displayed. Defaults to FALSE.
return_obj	Logical value. If TRUE (the default), an object of class GDALVector opened on the output layer will be returned, otherwise the function returns a logical value.

Details

Seven processing modes are available:

- Intersection: The output layer contains features whose geometries represent areas that are common between features in the input layer and in the method layer. The features in the output layer have attributes from both input and method layers.

- `Union`: The output layer contains features whose geometries represent areas that are either in the input layer, in the method layer, or in both. The features in the output layer have attributes from both input and method layers. For features which represent areas that are only in the input layer or only in the method layer the respective attributes have undefined values.
- `SymDifference`: The output layer contains features whose geometries represent areas that are in either in the input layer or in the method layer but not in both. The features in the output layer have attributes from both input and method layers. For features which represent areas that are only in the input or only in the method layer the respective attributes have undefined values.
- `Identity`: Identifies the features of the input layer with the ones from the method layer. The output layer contains features whose geometries represent areas that are in the input layer. The features in the output layer have attributes from both the input and method layers.
- `Update`: The update method creates a layer which adds features into the input layer from the method layer, possibly cutting features in the input layer. The features in the output layer have areas of the features of the method layer or those areas of the features of the input layer that are not covered by the method layer. The features of the output layer get their attributes from the input layer.
- `Clip`: The clip method creates a layer which has features from the input layer clipped to the areas of the features in the method layer. By default the output layer has attributes of the input layer.
- `Erase`: The erase method creates a layer which has features from the input layer whose areas are erased by the features in the method layer. By default, the output layer has attributes of the input layer.

By default, `ogr_proc()` will create the output layer with an empty schema. It will be initialized by GDAL to contain all fields in the input layer, or depending on the operation, all fields in both the input and method layers. In the latter case, the prefixes `"input_"` and `"method_"` will be added to the output field names by default. The default prefixes can be overridden in the `mode_opt` argument as described below.

Alternatively, the functions in the `ogr_manage` interface could be used to create an empty layer with user-defined schema (e.g., `ogr_ds_create()`, `ogr_layer_create()` and `ogr_field_create()`). If the schema of the output layer is set by the user and contains fields that have the same name as a field in both the input and method layers, then the attribute for an output feature will get the value from the feature of the method layer.

Options that affect processing can be set as NAME=VALUE pairs passed in the `mode_opt` argument. Some options are specific to certain processing modes as noted below:

- SKIP_FAILURES=YES/NO. Set it to YES to go on, even when a feature could not be inserted or a GEOS call failed.
- PROMOTE_TO_MULTI=YES/NO. Set to YES to convert Polygons into MultiPolygons, LineStrings to MultiLineStrings or Points to MultiPoints (only since GDAL 3.9.2 for the latter).
- INPUT_PREFIX=string. Set a prefix for the field names that will be created from the fields of the input layer.
- METHOD_PREFIX=string. Set a prefix for the field names that will be created from the fields of the method layer.

- USE_PREPARED_GEOMETRIES=YES/NO. Set to NO to not use prepared geometries to pretest intersection of features of method layer with features of input layer. Applies to `Intersection`, `Union`, `Identity`.
- PRETEST_CONTAINMENT=YES/NO. Set to YES to pretest the containment of features of method layer within the features of input layer. This will speed up the operation significantly in some cases. Requires that the prepared geometries are in effect. Applies to `Intersection`.
- KEEP_LOWER_DIMENSION_GEOMETRIES=YES/NO. Set to NO to skip result features with lower dimension geometry that would otherwise be added to the output layer. The default is YES, to add features with lower dimension geometry, but only if the result output has an UNKNOWN geometry type. Applies to `Intersection`, `Union`, `Identity`.

The input and method layers should have the same spatial reference system. No on-the-fly reprojection is done. When an output layer is created it will have the SRS of `input_lyr`.

Value

Upon successful completion, an object of class `GDALVector` is returned by default (`return_obj = TRUE`), or logical `TRUE` is returned if `return_obj = FALSE`. Logical `FALSE` is returned if an error occurs during processing.

Note

The first geometry field on a layer is always used.

For best performance use the minimum amount of features in the method layer and copy into a memory layer.

See Also

`GDALVector-class`, `ogr_define`, `ogr_manage`

Vector overlay operators:
https://en.wikipedia.org/wiki/Vector_overlay

Examples

```
# MTBS fires in Yellowstone National Park 1984-2022
dsn <- system.file("extdata/ynp_fires_1984_2022.gpkg", package="gdalraster")

# layer filtered to fires after 1988
lyr1 <- new(GDALVector, dsn, "mtbs_perims")
lyr1$setAttributeFilter("ig_year > 1988")
lyr1$getFeatureCount()

# second layer for the 1988 North Fork fire perimeter
sql <- "SELECT incid_name, ig_year, geom FROM mtbs_perims
        WHERE incid_name = 'NORTH FORK'"
lyr2 <- new(GDALVector, dsn, sql)
lyr2$getFeatureCount()

# intersect to obtain areas in the North Fork perimeter that have re-burned
tmp_dsn <- tempfile(fileext = ".gpkg")
```

```
opt <- c("INPUT_PREFIX=layer1_",
         "METHOD_PREFIX=layer2_",
         "PROMOTE_TO_MULTI=YES")

lyr_out <- ogr_proc(mode = "Intersection",
                    input_lyr = lyr1,
                    method_lyr = lyr2,
                    out_dsn = tmp_dsn,
                    out_lyr_name = "north_fork_reburned",
                    out_geom_type = "MULTIPOLYGON",
                    mode_opt = opt)

# the output layer has attributes of both the input and method layers
(d <- lyr_out$fetch(-1))

# clean up
lyr1$close()
lyr2$close()
lyr_out$close()
```

---

ogr_reproject *Reproject a vector layer*

Description

ogr_reproject() reprojects the features of a vector layer to a different spatial reference system and writes the new layer to a specified output dataset. The output may be in a different vector file format than the input dataset. A source SRS definition must be available in the source layer for reprojection to occur.

Usage

```
ogr_reproject(
  src_dsn,
  src_layer,
  out_dsn,
  out_srs,
  out_fmt = NULL,
  overwrite = FALSE,
  append = FALSE,
  nln = NULL,
  nlt = NULL,
  dsco = NULL,
  lco = NULL,
  dialect = NULL,
  spat_bbox = NULL,
  src_open_options = NULL,
  progress = FALSE,
```

```
  add_cl_arg = NULL,
  return_obj = TRUE
)
```

Arguments

src_dsn	Character string. The filename or database connection string specifying the vector data source containing the input layer.
src_layer	Character string. The name of the input layer in src_dsn to reproject. Optionally can be given as an SQL SELECT statement to be executed against src_dsn, defining the source layer as the result set. May also be given as empty string (""), in which case the first layer by index will be used (mainly useful for single-layer file formats such as ESRI Shapefile).
out_dsn	Character string. The filename or database connection string specifying the vector data source to which the output layer will be written.
out_srs	Character string specifying the output spatial reference system. May be in WKT format or any of the formats supported by srs_to_wkt().
out_fmt	Optional character string giving the GDAL short name of the output dataset format. Only used if out_dsn needs to be created. Generally can be NULL in which case the format will be guessed from the file extension.
overwrite	Logical value. TRUE to overwrite the output layer if it already exists. Defaults to FALSE.
append	Logical value. TRUE to append to the output layer if it already exists. Defaults to FALSE.
nln	Optional character string giving an alternate name to assign the new layer. By default, src_layer is used, but nln is required if src_layeris a SQL SELECT statement.
nlt	Optional character string to define the geometry type for the output layer. Mainly useful when nlt = PROMOTE_TO_MULTI might be given to automatically promote layers that mix polygon / multipolygons to multipolygons, and layers that mix linestrings / multilinestrings to multilinestrings. Can be useful when converting shapefiles to PostGIS and other output formats that implement strict checks for geometry types.
dsco	Optional character vector of format-specific creation options for out_dsn ("NAME=VALUE" pairs). Should only be used if out_dsn does not already exist.
lco	Optional character vector of format-specific creation options for the output layer ("NAME=VALUE" pairs). Should not be used if appending to an existing layer.
dialect	Optional character string specifying the SQL dialect to use. The OGR SQL engine ("OGRSQL") will be used by default if a value is not given. The "SQLite" dialect can also be use. Only relevant if src_layer is given as a SQL SELECT statement.
spat_bbox	Optional numeric vector of length four specifying a spatial bounding box (xmin, ymin, xmax, ymax), *in the SRS of the source layer*. Only features whose geometry intersects spat_bbox will be selected for reprojection.

src_open_options
: Optional character vector of dataset open options for src_dsn (format-specific "NAME=VALUE" pairs).

progress
: Logical value, TRUE to display progress on the terminal. Defaults to FALSE. Only works if the input layer has "fast feature count" capability.

add_cl_arg
: Optional character vector of additional command-line arguments to be passed to ogr2ogr() (see Note).

return_obj
: Logical value, TRUE to return an object of class GDALVector open on the output layer (the default).

Details

ogr_reproject() is a convenience wrapper to perform vector reprojection via ogr2ogr(), which in turn is an API binding to GDAL's ogr2ogr command-line utility.

Value

Upon successful completion, an object of class GDALVector is returned by default (if return_obj = TRUE), or logical TRUE is returned (invisibly) if return_obj = FALSE. An error is raised if reprojection fails.

Note

For advanced use, additional command-line arguments may be passed to ogr2ogr() in add_cl_arg (e.g., advanced geometry and SRS related options). Users should be aware of possible implications and compatibility with the arguments already implied by the parameterization of ogr_reproject().

The function will attempt to create the output datasource if it does not already exist. Some formats (e.g., PostgreSQL) do not support creation of new datasets (i.e., a database within PostgreSQL), but output layers can be written to an existing database.

See Also

ogr2ogr()

GDAL documentation for ogr2ogr:
https://gdal.org/en/stable/programs/ogr2ogr.html

warp() for raster reprojection

Examples

```
# MTBS fire perimeters
f <- system.file("extdata/ynp_fires_1984_2022.gpkg", package = "gdalraster")
(mtbs <- new(GDALVector, f, "mtbs_perims"))

mtbs$getSpatialRef() |> srs_is_projected()  # TRUE

# YNP boundary
f <- system.file("extdata/ynp_features.zip", package = "gdalraster")
unzip(f, files = "ynp_features.gpkg", exdir = tempdir())
ynp_dsn <- file.path(tempdir(), "ynp_features.gpkg")
```

```
(bnd <- new(GDALVector, ynp_dsn, "ynp_bnd"))

bnd$getSpatialRef() |> srs_is_projected()  # FALSE

# project the boundary to match the MTBS layer
out_dsn <- tempfile(fileext = ".gpkg")
(bnd_mtsp <- ogr_reproject(ynp_dsn, "ynp_bnd", out_dsn, mtbs$getSpatialRef()))

bnd_mtsp$getFeatureCount()

plot(bnd_mtsp$getNextFeature(), col = "wheat")

mtbs$setAttributeFilter("incid_name = 'MAPLE'")
mtbs$getFeatureCount()  # 1

(feat <- mtbs$getNextFeature())

plot(feat, col = "red", border = NA, add = TRUE)

mtbs$close()
bnd$close()
bnd_mtsp$close()
```

`pixel_extract` *Extract pixel values at geospatial point locations*

Description

`pixel_extract()` returns raster pixel values for a set of geospatial point locations. The coordinates are given as a two-column matrix of x/y values in the same spatial reference system as the input raster (unless `xy_srs` is specified). Coordinates can also be given in a data frame with an optional column of point IDs. Values are extracted from all bands of the raster by default, or specific band numbers may be given. An optional interpolation method may be specified for bilinear (2 x 2 kernel), cubic convolution (4 x 4 kernel, GDAL >= 3.10), or cubic spline (4 x 4 kernel, GDAL >= 3.10). Alternatively, an optional kernel dimension `N` may be given to extract values of the individual pixels within an `N x N` kernel centered on the pixel containing the point location. If `xy_srs` is given, the function will attempt to transform the input points to the projection of the raster with a call to `transform_xy()`.

Usage

```
pixel_extract(
  raster,
  xy,
  bands = NULL,
```

```
  interp = NULL,
  krnl_dim = NULL,
  xy_srs = NULL,
  max_ram = 300,
  as_data_frame = NULL
)
```

Arguments

raster	Either a character string giving the filename of a raster, or an object of class GDALRaster for the source dataset.
xy	A two-column numeric matrix or two-column data frame of geospatial coordinates (x, y), or a vector for a single point (x, y), in the same spatial reference system as raster. Can also be given as a data frame with three or more columns, in which the first column must be a vector of point IDs (character or numeric), and the second and third columns must contain the geospatial coordinates (x, y).
bands	Optional numeric vector of band numbers. All bands in raster will be processed by default if not specified, or if 0 is given.
interp	Optional character string specifying an interpolation method. Must be one of "bilinear", "cubic", "cubicspline", or "nearest" (the default if not specified, i.e., no interpolation). GDAL >= 3.10 is required for "cubic" and "cubicspline".
krnl_dim	Optional numeric value specifying the dimension N pixels of an N x N kernel for which all individual pixel values will be returned. Should be a positive whole number (will be coerced to integer by truncation). Currently only supported when extracting from a single raster band. Ignored if interp is specified as other than "nearest" (i.e., the kernel implied by the interpolation method will always be used).
xy_srs	Optional character string specifying the spatial reference system for xy. May be in WKT format or any of the formats supported by srs_to_wkt().
max_ram	Numeric value giving the maximum amount of RAM (in MB) to use for potentially copying a remote raster into memory for processing (see Note). Defaults to 300 MB. Set to zero to disable potential copy of remote files into memory.
as_data_frame	Logical value, TRUE to return output as a data frame. The default is to return a numeric matrix unless point IDs are present in the first column of xy given as a data frame. In that latter case, the output will always be a data frame with the point IDs in the first column.

Value

A numeric matrix or data frame of pixel values with number of rows equal to the number of rows in xy. The number of columns is equal to the number of bands (plus optional point ID column), or if krnl_dim = N is used, number of columns is equal to N * N (plus optional point ID column). Output is always a data frame if xy is given as a data frame with a column of point IDs (as_data_frame will be ignored in that case). Named columns indicate the band, e.g., "b1". If krnl_dim is used, named columns indicate band and pixel, e.g., "b1_p1", "b1_p2", ..., "b1_p9" if krnl_dim = 3. Pixels are in left-to-right, top-to-bottom order in the kernel.

Note

Depending on the number of input points, extracting from a raster on a remote filesystem may require a large number of HTTP range requests which may be slow (i.e., URLs/remote VSI filesystems). In that case, it may be faster to copy the raster into memory first (either as MEM format or to a /vsimem/ filesystem). `pixel_extract()` will attempt to automate that process if the total size of file(s) that would be copied does not exceed the threshold given by `max_ram`, and `nrow(xy) > 10` (requires GDAL >= 3.6).

For alternative workflows that involve copying to local storage, the data management functions (e.g., `copyDatasetFiles()`) and the VSI filesystem functions (e.g., `vsi_is_local()`, `vsi_stat()`, `vsi_copy_file()`) may be of interest.

Examples

```
pt_file <- system.file("extdata/storml_pts.csv", package="gdalraster")
# id, x, y in NAD83 / UTM zone 12N, same as the raster
pts <- read.csv(pt_file)
print(pts)

raster_file <- system.file("extdata/storml_elev.tif", package="gdalraster")

pixel_extract(raster_file, pts)

# or as GDALRaster object
ds <- new(GDALRaster, raster_file)
# optionally suppress progress reporting
# ds$quiet <- TRUE
pixel_extract(ds, pts)

# interpolated values
pixel_extract(raster_file, pts, interp = "bilinear")

# individual pixel values within a kernel
pixel_extract(raster_file, pts, krnl_dim = 3)

# lont/lat xy
pts_wgs84 <- transform_xy(pts[-1], srs_from = ds$getProjection(),
                          srs_to = "WGS84")

# transform the input xy
pixel_extract(ds, xy = pts_wgs84, xy_srs = "WGS84")

ds$close()
```

`plot.OGRFeature` *Plot the geometry of an* `OGRFeature` *object*

Description

Plot the geometry of an `OGRFeature` object

Usage

```
## S3 method for class 'OGRFeature'
plot(x, xlab = "x", ylab = "y", main = "", ...)
```

Arguments

x	An OGRFeature object.
xlab	Title for the x axis.
ylab	Title for the y axis.
main	The main title (on top).
...	Optional arguments passed to wk::wk_plot().

Value

The input, invisibly.

---

plot.OGRFeatureSet *Plot the geometry column of an* OGRFeatureSet

---

Description

Plot the geometry column of an OGRFeatureSet

Usage

```
## S3 method for class 'OGRFeatureSet'
plot(x, xlab = "x", ylab = "y", main = "", ...)
```

Arguments

x	An OGRFeatureSet.
xlab	Title for the x axis.
ylab	Title for the y axis.
main	The main title (on top).
...	Optional arguments passed to wk::wk_plot().

Value

The input, invisibly.

plot_geom *Plot WKT or WKB geometries*

Description

plot_geom() plots one or more geometries given as either WKT or WKB raw vectors, using wk::wk_plot().

Usage

```
plot_geom(x, xlab = "x", ylab = "y", main = "", ...)
```

Arguments

x	Either a character vector containing one or more WKT strings, a raw vector of WKB, or a list of WKB raw vectors.
xlab	Title for the x axis.
ylab	Title for the y axis.
main	The main title (on top).
...	Optional arguments passed to wk::wk_plot().

Value

The input, invisibly.

Examples

```
# a Delaunay triangulation of 10 random points
set.seed(4)
x <- sample.int(100, 10)
y <- sample.int(100, 10)

g <- g_create("MULTIPOINT", cbind(x, y))
g_wk2wk(g)

plot_geom(g)

g2 <- g_delaunay_triangulation(g)
g_wk2wk(g2)

plot_geom(g2, add = TRUE)
```

plot_raster *Display raster data*

Description

plot_raster() displays raster data using base graphics.

Usage

```
plot_raster(
  data,
  xsize = NULL,
  ysize = NULL,
  nbands = NULL,
  max_pixels = 2.5e+07,
  col_tbl = NULL,
  maxColorValue = 1,
  normalize = TRUE,
  minmax_def = NULL,
  minmax_pct_cut = NULL,
  col_map_fn = NULL,
  pixel_fn = NULL,
  xlim = NULL,
  ylim = NULL,
  interpolate = TRUE,
  asp = 1,
  axes = TRUE,
  main = "",
  xlab = "x",
  ylab = "y",
  xaxs = "i",
  yaxs = "i",
  legend = FALSE,
  digits = 2,
  na_col = rgb(0, 0, 0, 0),
  ...
)
```

Arguments

data	Either a GDALRaster object from which data will be read, or a numeric vector of pixel values arranged in left to right, top to bottom order, or a list of band vectors. If input is vector or list, the information in attribute gis will be used if present (see read_ds()), potentially ignoring values below for xsize, ysize, nbands.
xsize	The number of pixels along the x dimension in data. If data is a GDALRaster object, specifies the size at which the raster will be read (used for argument

	out_xsize in GDALRaster$read()). By default, the entire raster will be read at full resolution.
ysize	The number of pixels along the y dimension in data. If data is a GDALRaster object, specifies the size at which the raster will be read (used for argument out_ysize in GDALRaster$read()). By default, the entire raster will be read at full resolution.
nbands	The number of bands in data. Must be either 1 (grayscale) or 3 (RGB). For RGB, data are interleaved by band. If nbands is NULL (the default), then nbands = 3 is assumed if the input data contain 3 bands, otherwise band 1 is used.
max_pixels	The maximum number of pixels that the function will attempt to display (per band). An error is raised if (xsize * ysize) exceeds this value. Setting to NULL turns off this check.
col_tbl	A color table as a matrix or data frame with four or five columns. Column 1 contains the numeric pixel values. Columns 2:4 contain the intensities of the red, green and blue primaries (0:1 by default, or use integer 0:255 by setting maxColorValue = 255). An optional column 5 may contain alpha transparency values, 0 for fully transparent to 1 (or maxColorValue) for opaque (the default if column 5 is missing). If data is a GDALRaster object, a built-in color table will be used automatically if one exists in the dataset.
maxColorValue	A number giving the maximum of the color values range in col_tbl (see above). The default is 1.
normalize	Logical. TRUE to rescale pixel values so that their range is [0,1], normalized to the full range of the pixel data by default (min(data), max(data), per band). Ignored if col_tbl is used. Set normalize to FALSE if a color map function is used that operates on raw pixel values (see col_map_fn below).
minmax_def	Normalize to user-defined min/max values (in terms of the pixel data, per band). For single-band grayscale, a numeric vector of length two containing min, max. For 3-band RGB, a numeric vector of length six containing b1_min, b2_min, b3_min, b1_max, b2_max, b3_max.
minmax_pct_cut	Normalize to a truncated range of the pixel data using percentile cutoffs (removes outliers). A numeric vector of length two giving the percentiles to use (e.g., c(2, 98)). Applied per band. Ignored if minmax_def is used.
col_map_fn	An optional color map function (default is grDevices::gray for single-band data or grDevices::rgb for 3-band). Ignored if col_tbl is used. Set normalize to FALSE if using a color map function that operates on raw pixel values.
pixel_fn	An optional function that will be applied to the input pixel data. Must accept vector input and return a numeric vector of the same length as its input.
xlim	Numeric vector of length two giving the x coordinate range. If data is a GDALRaster object, the default is the raster xmin, xmax in georeferenced coordinates, otherwise the default uses pixel/line coordinates (c(0, xsize)).
ylim	Numeric vector of length two giving the y coordinate range. If data is a GDALRaster object, the default is the raster ymin, ymax in georeferenced coordinates, otherwise the default uses pixel/line coordinates (c(ysize, 0)).
interpolate	Logical indicating whether to apply linear interpolation to the image when drawing (default TRUE).

asp	Numeric. The aspect ratio y/x (see ?plot.window).
axes	Logical. TRUE to draw axes (the default).
main	The main title (on top).
xlab	Title for the x axis (see ?title).
ylab	Title for the y axis (see ?title).
xaxs	The style of axis interval calculation to be used for the x axis (see ?par).
yaxs	The style of axis interval calculation to be used for the y axis (see ?par).
legend	Logical indicating whether to include a legend on the plot. Currently, legends are only supported for continuous data. A color table will be used if one is specified or the raster has a built-in color table, otherwise the value for col_map_fn will be used.
digits	The number of digits to display after the decimal point in the legend labels when raster data are floating point.
na_col	Color to use for NA as a 7- or 9-character hexadecimal code. The default is transparent ("#00000000", the return value of rgb(0,0,0,0)).
...	Other parameters to be passed to plot.default().

Details

By default, contrast enhancement by stretch to min/max is applied when the input data are single-band grayscale with any raster data type, or three-band RGB with raster data type larger than Byte. The minimum/maximum of the input data are used by default (i.e., no outlier removal). No stretch is applied by default when the input is an RGB byte raster. These defaults can be overridden by specifying either the minmax_def argument (user-defined min/max per band), or the minmax_pct_cut argument (ignore outlier pixels based on a percentile range per band). These settings (and the normalize argument) are ignored if a color table is used.

Note

plot_raster() uses the function graphics::rasterImage() for plotting which is not supported on some devices (see ?rasterImage).

If data is an object of class GDALRaster, then plot_raster() will attempt to read the entire raster into memory by default (unless the number of pixels per band would exceed max_pixels). A reduced resolution overview can be read by setting xsize, ysize smaller than the raster size on disk. (If data is instead specified as a vector of pixel values, a reduced resolution overview would be read by setting out_xsize and out_ysize smaller than the raster region defined by xsize, ysize in a call to GDALRaster$read()). The GDAL_RASTERIO_RESAMPLING configuration option can be defined to override the default resampling (NEAREST) to one of BILINEAR, CUBIC, CUBICSPLINE, LANCZOS, AVERAGE or MODE, for example:

```
set_config_option("GDAL_RASTERIO_RESAMPLING", "BILINEAR")
```

See Also

GDALRaster$read(), read_ds(), set_config_option()

Examples

```
## Elevation
elev_file <- system.file("extdata/storml_elev.tif", package="gdalraster")
ds <- new(GDALRaster, elev_file)

# all other arguments are optional when passing a GDALRaster object
# grayscale
plot_raster(ds, legend = TRUE, main = "Storm Lake elevation (m)")

# color ramp from user-defined palette
elev_pal <- c("#00A60E","#63C600","#E6E600","#E9BD3B",
              "#ECB176","#EFC2B3","#F2F2F2")
ramp <- scales::colour_ramp(elev_pal, alpha = FALSE)
plot_raster(ds, col_map_fn = ramp, legend = TRUE,
            main = "Storm Lake elevation (m)")

ds$close()

## Landsat band combination
b4_file <- system.file("extdata/sr_b4_20200829.tif", package="gdalraster")
b5_file <- system.file("extdata/sr_b5_20200829.tif", package="gdalraster")
b6_file <- system.file("extdata/sr_b6_20200829.tif", package="gdalraster")
band_files <- c(b6_file, b5_file, b4_file)

vrt_file <- file.path(tempdir(), "storml_b6_b5_b4.vrt")
buildVRT(vrt_file, band_files, cl_arg = "-separate")

ds <- new(GDALRaster, vrt_file)

plot_raster(ds, main = "Landsat 6-5-4 (vegetative analysis)")

ds$close()


## LANDFIRE Existing Vegetation Cover (EVC) with color map
evc_file <- system.file("extdata/storml_evc.tif", package="gdalraster")

# colors from the CSV attribute table distributed by LANDFIRE
evc_csv <- system.file("extdata/LF20_EVC_220.csv", package="gdalraster")
vat <- read.csv(evc_csv)
head(vat)
vat <- vat[, c(1, 6:8)]

ds <- new(GDALRaster, evc_file)
plot_raster(ds, col_tbl = vat, interpolate = FALSE,
            main = "Storm Lake LANDFIRE EVC")

ds$close()

## Apply a pixel function
f <- system.file("extdata/complex.tif", package="gdalraster")
ds <- new(GDALRaster, f)
```

```
ds$getDataTypeName(band = 1)  # complex floating point

ramp <- scales::colour_ramp(scales::pal_viridis(option = "plasma")(6),
                            alpha = FALSE)

plot_raster(ds, pixel_fn = Arg, col_map_fn = ramp, interpolate = FALSE,
            legend = TRUE, main = "Arg(complex.tif)")

ds$close()
```

`polygonize` *Create a polygon feature layer from raster data*

Description

`polygonize()` creates vector polygons for all connected regions of pixels in a source raster sharing a common pixel value. Each polygon is created with an attribute indicating the pixel value of that polygon. A raster mask may also be provided to determine which pixels are eligible for processing. The function will create the output vector layer if it does not already exist, otherwise it will try to append to an existing one. This function is a wrapper of `GDALPolygonize` in the GDAL Algorithms API. It provides essentially the same functionality as the `gdal_polygonize.py` command-line program (https://gdal.org/en/stable/programs/gdal_polygonize.html).

Usage

```
polygonize(
  raster_file,
  out_dsn,
  out_layer,
  fld_name = "DN",
  out_fmt = NULL,
  connectedness = 4,
  src_band = 1,
  mask_file = NULL,
  nomask = FALSE,
  overwrite = FALSE,
  dsco = NULL,
  lco = NULL,
  quiet = FALSE
)
```

Arguments

`raster_file`	Filename of the source raster.
`out_dsn`	The destination vector filename to which the polygons will be written (or database connection string).

out_layer	Name of the layer for writing the polygon features. For single-layer file formats such as "ESRI Shapefile", the layer name is the same as the filename without the path or extension (e.g., out_dsn = "path_to_file/polygon_output.shp", the layer name is "polygon_output").
fld_name	Name of an integer attribute field in out_layer to which the pixel values will be written. Will be created if necessary when using an existing layer.
out_fmt	GDAL short name of the output vector format. If unspecified, the function will attempt to guess the format from the filename/connection string.
connectedness	Integer scalar. Must be either 4 or 8. For the default 4-connectedness, pixels with the same value are considered connected only if they touch along one of the four sides, while 8-connectedness also includes pixels that touch at one of the corners.
src_band	The band on raster_file to build the polygons from (default is 1).
mask_file	Use the first band of the specified raster as a validity mask (zero is invalid, non-zero is valid). If not specified, the default validity mask for the input band (such as nodata, or alpha masks) will be used (unless nomask is set to TRUE).
nomask	Logical scalar. If TRUE, do not use the default validity mask for the input band (such as nodata, or alpha masks). Default is FALSE.
overwrite	Logical scalar. If TRUE, overwrite out_layer if it already exists. Default is FALSE.
dsco	Optional character vector of format-specific creation options for out_dsn ("NAME=VALUE" pairs).
lco	Optional character vector of format-specific creation options for out_layer ("NAME=VALUE" pairs).
quiet	Logical scalar. If TRUE, a progress bar will not be displayed. Defaults to FALSE.

Details

Polygon features will be created on the output layer, with polygon geometries representing the polygons. The polygon geometries will be in the georeferenced coordinate system of the raster (based on the geotransform of the source dataset). It is acceptable for the output layer to already have features. If the output layer does not already exist, it will be created with coordinate system matching the source raster.

The algorithm attempts to minimize memory use so that very large rasters can be processed. However, if the raster has many polygons or very large/complex polygons, the memory use for holding polygon enumerations and active polygon geometries may grow to be quite large.

The algorithm will generally produce very dense polygon geometries, with edges that follow exactly on pixel boundaries for all non-interior pixels. For non-thematic raster data (such as satellite images) the result will essentially be one small polygon per pixel, and memory and output layer sizes will be substantial. The algorithm is primarily intended for relatively simple thematic rasters, masks, and classification results.

Note

The source pixel band values are read into a signed 64-bit integer buffer (Int64) by GDALPolygonize, so floating point or complex bands will be implicitly truncated before processing.

When 8-connectedness is used, many of the resulting polygons will likely be invalid due to ring self-intersection (in the strict OGC definition of polygon validity). See `g_is_valid()` / `g_make_valid()` (single polygons can become MultiPolygon' in the case of self-intersections).

If writing to a SQLite database format as either `GPKG` (GeoPackage vector) or `SQLite` (Spatialite vector), setting the `SQLITE_USE_OGR_VFS` and `OGR_SQLITE_JOURNAL` configuration options may increase performance substantially. If writing to `PostgreSQL` (PostGIS vector), setting `PG_USE_COPY=YES` is faster:

```
# SQLite: GPKG (.gpkg) and Spatialite (.sqlite)
# enable extra buffering/caching by the GDAL/OGR I/O layer
set_config_option("SQLITE_USE_OGR_VFS", "YES")
# set the journal mode for the SQLite database to MEMORY
set_config_option("OGR_SQLITE_JOURNAL", "MEMORY")

# PostgreSQL / PostGIS
# use COPY for inserting data rather than INSERT
set_config_option("PG_USE_COPY", "YES")
```

See Also

`rasterize()`

`vignette("gdal-config-quick-ref")`

Examples

```
evt_file <- system.file("extdata/storml_evt.tif", package="gdalraster")
dsn <- file.path(tempdir(), "storm_lake.gpkg")
layer <- "lf_evt"
fld <- "evt_value"
set_config_option("SQLITE_USE_OGR_VFS", "YES")
set_config_option("OGR_SQLITE_JOURNAL", "MEMORY")
polygonize(evt_file, dsn, layer, fld)
set_config_option("SQLITE_USE_OGR_VFS", "")
set_config_option("OGR_SQLITE_JOURNAL", "")
```

---

pop_error_handler *Pop error handler off stack*

Description

`pop_error_handler()` is a wrapper for `CPLPopErrorHandler()` in the GDAL Common Portability Library. Discards the current error handler on the error handler stack, and restores the one in use before the last `push_error_handler()` call. This method has no effect if there are no error handlers on the current thread's error handler stack.

Usage

```
pop_error_handler()
```

Value

No return value, called for side effects.

See Also

push_error_handler()

Examples

```
push_error_handler("quiet")
# ...
pop_error_handler()
```

---

print.OGRFeature *Print an* OGRFeature *object*

Description

Print an OGRFeature object

Usage

```
## S3 method for class 'OGRFeature'
print(x, ...)
```

Arguments

x	An OGRFeature object.
...	Optional arguments passed to base::print().

Value

The input, invisibly.

print.OGRFeatureSet *Print an* OGRFeatureSet

Description

Print an OGRFeatureSet

Usage

```
## S3 method for class 'OGRFeatureSet'
print(x, ...)
```

Arguments

x	An OGRFeatureSet.
...	Optional arguments passed to base::print.data.frame().

Value

The input, invisibly.

proj_networking *Check, enable or disable PROJ networking capabilities*

Description

proj_networking() returns the status of PROJ networking capabilities, optionally enabling or disabling first. Requires GDAL 3.4 or later and PROJ 7 or later.

Usage

```
proj_networking(enabled = NULL)
```

Arguments

enabled	Optional logical scalar. Set to TRUE to enable networking capabilities or FALSE to disable.

Value

Logical TRUE if PROJ networking capabilities are enabled (as indicated by the return value of OSRGetPROJEnableNetwork() in the GDAL Spatial Reference System C API). Logical NA is returned if GDAL < 3.4.

See Also

proj_version(), proj_search_paths()

PROJ-data on GitHub, PROJ Content Delivery Network

Examples

```
proj_networking()
```

---

proj_search_paths *Get or set search path(s) for PROJ resource files*

Description

proj_search_paths() returns the search path(s) for PROJ resource files, optionally setting them first.

Usage

```
proj_search_paths(paths = NULL)
```

Arguments

paths	Optional character vector containing one or more directory paths to set.

Value

A character vector containing the currently used search path(s) for PROJ resource files. An empty string ("") is returned if no search paths are returned by the function OSRGetPROJSearchPaths() in the GDAL Spatial Reference System C API.

See Also

proj_version(), proj_networking()

Examples

```
proj_search_paths()
```

proj_version *Get PROJ version*

Description

`proj_version()` returns version information for the PROJ library in use by GDAL.

Usage

```
proj_version()
```

Value

A list of length four containing:

- `name` - a string formatted as "major.minor.patch"
- `major` - major version as integer
- `minor` - minor version as integer
- `patch` - patch version as integer

See Also

`gdal_version()`, `geos_version()`, `proj_search_paths()`, `proj_networking()`

Examples

```
proj_version()
```

push_error_handler *Push a new GDAL CPLError handler*

Description

`push_error_handler()` is a wrapper for `CPLPushErrorHandler()` in the GDAL Common Portability Library. This pushes a new error handler on the thread-local error handler stack. This handler will be used until removed with `pop_error_handler()`. A typical use is to temporarily set `CPLQuietErrorHandler()` which doesn't make any attempt to report passed error or warning messages, but will process debug messages via `CPLDefaultErrorHandler`.

Usage

```
push_error_handler(handler)
```

Arguments

`handler`	Character name of the error handler to push. One of `quiet`, `logging` or `default`.

Value

No return value, called for side effects.

Note

This function is for advanced use cases. It is intended for setting a *temporary* error handler in a limited segment of code. A global error handler specific to the R environment is in use by default.

Setting `handler = "logging"` will use `CPLLoggingErrorHandler()`, error handler that logs into the file defined by the `CPL_LOG` configuration option. Be sure that option is set when using this error handler.

This only affects error reporting from GDAL.

See Also

`pop_error_handler()`

Examples

```
push_error_handler("quiet")
# ...
pop_error_handler()
```

---

rasterFromRaster *Create a raster from an existing raster as template*

Description

`rasterFromRaster()` creates a new raster with spatial reference, extent and resolution taken from a template raster, without copying data. Optionally changes the format, number of bands, data type and nodata value, sets driver-specific dataset creation options, and initializes to a value.

Usage

```
rasterFromRaster(
  srcfile,
  dstfile,
  fmt = NULL,
  nbands = NULL,
  dtName = NULL,
  options = NULL,
  init = NULL,
  dstnodata = init
)
```

Arguments

srcfile	Source raster filename.
dstfile	Output raster filename.
fmt	Output raster format name (e.g., "GTiff" or "HFA"). Will attempt to guess from the output filename if fmt is not specified.
nbands	Number of output bands.
dtName	Output raster data type name. Commonly used types include "Byte", "Int16", "UInt16", "Int32" and "Float32".
options	Optional list of format-specific creation options in a vector of "NAME=VALUE" pairs (e.g., options = c("COMPRESS=LZW") to set LZW compression during creation of a GTiff file).
init	Numeric value to initialize all pixels in the output raster.
dstnodata	Numeric nodata value for the output raster.

Value

Returns the destination filename invisibly.

See Also

GDALRaster-class, create(), createCopy(), bandCopyWholeRaster(), translate()

Examples

```
# band 2 in a FARSITE landscape file has slope degrees
# convert slope degrees to slope percent in a new raster
lcp_file <- system.file("extdata/storm_lake.lcp", package="gdalraster")
ds_lcp <- new(GDALRaster, lcp_file)
ds_lcp$getMetadata(band = 2, domain = "")

slpp_file <- file.path(tempdir(), "storml_slpp.tif")
opt = c("COMPRESS=LZW")
rasterFromRaster(srcfile = lcp_file,
                 dstfile = slpp_file,
                 nbands = 1,
                 dtName = "Int16",
                 options = opt,
                 init = -32767)
ds_slp <- new(GDALRaster, slpp_file, read_only = FALSE)

# slpp_file is initialized to -32767 and nodata value set
ds_slp$getNoDataValue(band=1)

# extent and cell size are the same as lcp_file
ds_lcp$bbox()
ds_lcp$res()
ds_slp$bbox()
ds_slp$res()
```

```
# convert slope degrees in lcp_file band 2 to slope percent in slpp_file
# bring through LCP nodata -9999 to the output nodata value
ncols <- ds_slp$getRasterXSize()
nrows <- ds_slp$getRasterYSize()
for (row in 0:(nrows-1)) {
    rowdata <- ds_lcp$read(band = 2,
                           xoff = 0, yoff = row,
                           xsize = ncols, ysize = 1,
                           out_xsize = ncols, out_ysize = 1)
    rowslpp <- tan(rowdata*pi/180) * 100
    rowslpp[rowdata==-9999] <- -32767
    dim(rowslpp) <- c(1, ncols)
    ds_slp$write(band = 1, xoff = 0, yoff = row,
                 xsize = ncols, ysize = 1,
                 rasterData = rowslpp)
}

# min, max, mean, sd
ds_slp$getStatistics(band = 1, approx_ok = FALSE, force = TRUE)

ds_slp$close()
ds_lcp$close()
```

`rasterize` *Burn vector geometries into a raster*

Description

`rasterize()` burns vector geometries (points, lines, or polygons) into the band(s) of a raster dataset. Vectors are read from any GDAL OGR-supported vector format. This function is a wrapper for the `gdal_rasterize` command-line utility (https://gdal.org/en/stable/programs/gdal_rasterize.html).

Usage

```
rasterize(
  src_dsn,
  dstfile,
  band = NULL,
  layer = NULL,
  where = NULL,
  sql = NULL,
  burn_value = NULL,
  burn_attr = NULL,
  invert = NULL,
  te = NULL,
  tr = NULL,
  tap = NULL,
```

```
  ts = NULL,
  dtName = NULL,
  dstnodata = NULL,
  init = NULL,
  fmt = NULL,
  co = NULL,
  add_options = NULL,
  quiet = FALSE
)
```

Arguments

src_dsn	Data source name for the input vector layer (filename or connection string).
dstfile	Either a character string giving the filename of the output raster dataset, or an object of class GDALRaster for the output. Must support update mode access. If given as a filename, this file will be created (or overwritten if it already exists - see Note). If given as a GDALRaster object for an existing dataset, then the affected pixels are updated in-place (object must be open with write access).
band	Numeric vector. The band(s) to burn values into (for existing dstfile). The default is to burn into band 1. Not used when creating a new raster.
layer	Character vector of layer names(s) from src_dsn that will be used for input features. At least one layer name or a sql option must be specified.
where	An optional SQL WHERE style query string to select features to burn in from the input layer(s).
sql	An SQL statement to be evaluated against src_dsn to produce a virtual layer of features to be burned in (alternative to layer).
burn_value	A fixed numeric value to burn into a band for all features. A numeric vector can be supplied, one burn value per band being written to.
burn_attr	Character string. Name of an attribute field on the features to be used for a burn-in value. The value will be burned into all output bands.
invert	Logical scalar. TRUE to invert rasterization. Burn the fixed burn value, or the burn value associated with the first feature, into all parts of the raster not inside the provided polygon.
te	Numeric vector of length four. Sets the output raster extent. The values must be expressed in georeferenced units. If not specified, the extent of the output raster will be the extent of the vector layer.
tr	Numeric vector of length two. Sets the target pixel resolution. The values must be expressed in georeferenced units. Both must be positive.
tap	Logical scalar. (target aligned pixels) Align the coordinates of the extent of the output raster to the values of tr, such that the aligned extent includes the minimum extent. Alignment means that xmin / resx, ymin / resy, xmax / resx and ymax / resy are integer values.
ts	Numeric vector of length two. Sets the output raster size in pixels (xsize, ysize). Note that ts cannot be used with tr.

dtName	Character name of output raster data type, e.g., Byte, Int16, UInt16, Int32, UInt32, Float32, Float64. Defaults to Float64.
dstnodata	Numeric scalar. Assign a nodata value to output bands.
init	Numeric vector. Pre-initialize the output raster band(s) with these value(s). However, it is not marked as the nodata value in the output file. If only one value is given, the same value is used in all the bands.
fmt	Output raster format short name (e.g., "GTiff"). Will attempt to guess from the output filename if fmt is not specified.
co	Optional list of format-specific creation options for the output raster in a vector of "NAME=VALUE" pairs (e.g., options = c("TILED=YES","COMPRESS=LZW") to set LZW compression during creation of a tiled GTiff file).
add_options	An optional character vector of additional command-line options to gdal_rasterize (see the gdal_rasterize documentation at the URL above for all available options).
quiet	Logical scalar. If TRUE, a progress bar will not be displayed. Defaults to FALSE.

Value

Logical indicating success (invisible TRUE). An error is raised if the operation fails.

Note

rasterize() creates a new target raster when dstfile is given as a filename (character string). In that case, some combination of the fmt, dstnodata, init, co, te, tr, tap, ts, and dtName arguments will be used. The resolution or size must be specified using either the tr or ts argument for all new rasters. The target raster will be overwritten if it already exists and any of these creation-related options are used.

To update an existing raster in-place, an object of class GDALRaster may be given for the dstfile argument. The GDALRaster object should be open for write access.

See Also

polygonize()

Examples

```
# MTBS fire perimeters for Yellowstone National Park 1984-2022
dsn <- system.file("extdata/ynp_fires_1984_2022.gpkg", package="gdalraster")
sql <- "SELECT * FROM mtbs_perims ORDER BY mtbs_perims.ig_year"
out_file <- file.path(tempdir(), "ynp_fires_1984_2022.tif")

rasterize(src_dsn = dsn,
          dstfile = out_file,
          sql = sql,
          burn_attr = "ig_year",
          tr = c(90,90),
          tap = TRUE,
          dtName = "Int16",
```

```
            dstnodata = -9999,
            init = -9999,
            co = c("TILED=YES", "COMPRESS=LZW"))

ds <- new(GDALRaster, out_file)
pal <- scales::viridis_pal(end = 0.8, direction = -1)(6)
ramp <- scales::colour_ramp(pal)
plot_raster(ds, legend = TRUE, col_map_fn = ramp, na_col = "#d9d9d9",
            main = "YNP Fires 1984-2022 - Most Recent Burn Year")

ds$close()
```

`rasterToVRT` *Create a GDAL virtual raster derived from one source dataset*

Description

`rasterToVRT()` creates a virtual raster dataset (VRT format) derived from one source dataset with options for virtual subsetting, virtually resampling the source data at a different pixel resolution, or applying a virtual kernel filter. (See `buildVRT()` for virtual mosaicing.)

Usage

```
rasterToVRT(
  srcfile,
  relativeToVRT = FALSE,
  vrtfile = tempfile("tmprast", fileext = ".vrt"),
  resolution = NULL,
  subwindow = NULL,
  src_align = TRUE,
  resampling = "nearest",
  krnl = NULL,
  normalized = TRUE,
  krnl_fn = NULL
)
```

Arguments

`srcfile`	Source raster filename.
`relativeToVRT`	Logical. Indicates whether the source filename should be interpreted as relative to the .vrt file (`TRUE`) or not relative to the .vrt file (`FALSE`, the default). If `TRUE`, the .vrt file is assumed to be in the same directory as `srcfile` and `basename(srcfile)` is used in the .vrt file. Use `TRUE` if the .vrt file will always be stored in the same directory with `srcfile`.
`vrtfile`	Output VRT filename.

`resolution` A numeric vector of length two (xres, yres). The pixel size must be expressed in georeferenced units. Both must be positive values. The source pixel size is used if `resolution` is not specified.

`subwindow` A numeric vector of length four (xmin, ymin, xmax, ymax). Selects `subwindow` of the source raster with corners given in georeferenced coordinates (in the source CRS). If not given, the upper left corner of the VRT will be the same as source, and the VRT extent will be the same or larger than source depending on `resolution`.

`src_align` Logical.

- `TRUE`: the upper left corner of the VRT extent will be set to the upper left corner of the source pixel that contains `subwindow` xmin, ymax. The VRT will be pixel-aligned with source if the VRT `resolution` is the same as the source pixel size, otherwise VRT extent will be the minimum rectangle that contains `subwindow` for the given pixel size. Often, `src_align=TRUE` when selecting a raster minimum bounding box for a vector polygon.
- `FALSE`: the VRT upper left corner will be exactly `subwindow` xmin, ymax, and the VRT extent will be the minimum rectangle that contains `subwindow` for the given pixel size. If `subwindow` is not given, the source raster extent is used in which case `src_align=FALSE` has no effect. Use `src_align=FALSE` to pixel-align two rasters of different sizes, i.e., when the intent is target alignment.

`resampling` The resampling method to use if xsize, ysize of the VRT is different than the size of the underlying source rectangle (in number of pixels). The values allowed are nearest, bilinear, cubic, cubicspline, lanczos, average and mode (as character).

`krnl` A filtering kernel specified as pixel coefficients. `krnl` is a array with dimensions (size, size), where size must be an odd number. `krnl` can also be given as a vector with length size x size. For example, a 3x3 average filter is given by:

```
krnl <- c(0.11111, 0.11111, 0.11111,
          0.11111, 0.11111, 0.11111,
          0.11111, 0.11111, 0.11111)
```

A kernel cannot be applied to sub-sampled or over-sampled data.

`normalized` Logical. Indicates whether the kernel is normalized. Defaults to `TRUE`.

`krnl_fn` Character string specifying a function to compute on the given `krnl`. Must be one of `"min"`, `"max"`, `"stddev"`, `"median"` or `"mode"`. *Requires GDAL >= 3.12*. E.g., to compute the median value in a 3x3 neighborhood around each pixel:

```
krnl <- c(1, 1, 1,
          1, 1, 1,
          1, 1, 1)

krnl_fn <- "median"
```

Details

`rasterToVRT()` can be used to virtually clip and pixel-align various raster layers with each other or in relation to vector polygon boundaries. It also supports VRT kernel filtering.

A VRT dataset is saved as a plain-text file with extension .vrt. This file contains a description of the dataset in an XML format. The description includes the source raster filename which can be a full path (`relativeToVRT = FALSE`) or relative path (`relativeToVRT = TRUE`). For relative path, `rasterToVRT()` assumes that the .vrt file will be in the same directory as the source file and uses `basename(srcfile)`. The elements of the XML schema describe how the source data will be read, along with algorithms potentially applied and so forth. Documentation of the XML format for .vrt is at: https://gdal.org/en/stable/drivers/raster/vrt.html.

Since .vrt is a small plain-text file it is fast to write and requires little storage space. Read performance is not degraded for certain simple operations (e.g., virtual clip without resampling). Reading will be slower for virtual resampling to a different pixel resolution or virtual kernel filtering since the operations are performed on-the-fly (but .vrt does not require the up front writing of a resampled or kernel-filtered raster to a regular format). VRT is sometimes useful as an intermediate raster in a series of processing steps, e.g., as a `tempfile` (the default).

GDAL VRT format has several capabilities and uses beyond those covered by `rasterToVRT()`. See the URL above for a full discussion.

Value

Returns the VRT filename invisibly.

Note

Pixel alignment is specified in terms of the source raster pixels (i.e., `srcfile` of the virtual raster). The use case in mind is virtually clipping a raster to the bounding box of a vector polygon and keeping pixels aligned with `srcfile` (`src_align = TRUE`). `src_align` would be set to `FALSE` if the intent is "target alignment". For example, if `subwindow` is the bounding box of another raster with a different layout, then also setting `resolution` to the pixel resolution of the target raster and `src_align = FALSE` will result in a virtual raster pixel-aligned with the target (i.e., pixels in the virtual raster are no longer aligned with its `srcfile`). Resampling defaults to `nearest` if not specified. Examples for both cases of `src_align` are given below.

`rasterToVRT()` assumes `srcfile` is a north-up raster.

See Also

`GDALRaster-class`, `bbox_from_wkt()`, `buildVRT()`

`warp()` can write VRT for virtual reprojection

Examples

```
## resample

evt_file <- system.file("extdata/storml_evt.tif", package="gdalraster")
ds <- new(GDALRaster, evt_file)
ds$res()
ds$bbox()
ds$close()

# table of the unique pixel values and their counts
tbl <- buildRAT(evt_file)
```

```
print(tbl)
sum(tbl$COUNT)

# resample at 90-m resolution
# EVT is thematic vegetation type so use a majority value
vrt_file <- rasterToVRT(evt_file,
                        resolution = c(90, 90),
                        resampling = "mode")

# .vrt is a small xml file pointing to the source raster
file.size(vrt_file)

tbl90m <- buildRAT(vrt_file)
print(tbl90m)
sum(tbl90m$COUNT)

ds <- new(GDALRaster, vrt_file)
ds$res()
ds$bbox()
ds$close()


## clip

evt_file <- system.file("extdata/storml_evt.tif", package="gdalraster")
ds_evt <- new(GDALRaster, evt_file)
ds_evt$bbox()

# WKT string for a boundary within the EVT extent
bnd = "POLYGON ((324467.3 5104814.2, 323909.4 5104365.4, 323794.2
5103455.8, 324970.7 5102885.8, 326420.0 5103595.3, 326389.6 5104747.5,
325298.1 5104929.4, 325298.1 5104929.4, 324467.3 5104814.2))"

# src_align = TRUE
vrt_file <- rasterToVRT(evt_file,
                        subwindow = bbox_from_wkt(bnd),
                        src_align = TRUE)
ds_vrt <- new(GDALRaster, vrt_file)

# VRT is a virtual clip, pixel-aligned with the EVT raster
bbox_from_wkt(bnd)
ds_vrt$bbox()
ds_vrt$res()
ds_vrt$close()


# src_align = FALSE
vrt_file <- rasterToVRT(evt_file,
                        subwindow = bbox_from_wkt(bnd),
                        src_align = FALSE)
ds_vrt_noalign <- new(GDALRaster, vrt_file)

# VRT upper left corner (xmin, ymax) is exactly bnd xmin, ymax
```

```
ds_vrt_noalign$bbox()
ds_vrt_noalign$res()

ds_vrt_noalign$close()
ds_evt$close()



## subset and pixel align two rasters

# FARSITE landscape file for the Storm Lake area
lcp_file <- system.file("extdata/storm_lake.lcp", package="gdalraster")
ds_lcp <- new(GDALRaster, lcp_file)

# Landsat band 5 file covering the Storm Lake area
b5_file <- system.file("extdata/sr_b5_20200829.tif", package="gdalraster")
ds_b5 <- new(GDALRaster, b5_file)

ds_lcp$bbox()  # 323476.1 5101872.0  327766.1 5105082.0
ds_lcp$res()   # 30 30

ds_b5$bbox()   # 323400.9 5101815.8  327870.9 5105175.8
ds_b5$res()    # 30 30

# src_align = FALSE because we need target alignment in this case:
vrt_file <- rasterToVRT(b5_file,
                        resolution = ds_lcp$res(),
                        subwindow = ds_lcp$bbox(),
                        src_align = FALSE)
ds_b5vrt <- new(GDALRaster, vrt_file)

ds_b5vrt$bbox() # 323476.1 5101872.0  327766.1 5105082.0
ds_b5vrt$res()  # 30 30

# read the the Landsat file pixel-aligned with the LCP file
# summarize band 5 reflectance where FBFM = 165
# LCP band 4 contains FBFM (a classification of fuel beds):
ds_lcp$getMetadata(band = 4, domain = "")

# verify Landsat nodata (0):
ds_b5vrt$getNoDataValue(band=1)
# will be read as NA and omitted from stats
rs <- new(RunningStats, na_rm = TRUE)

ncols <- ds_lcp$getRasterXSize()
nrows <- ds_lcp$getRasterYSize()
for (row in 0:(nrows-1)) {
    row_fbfm <- ds_lcp$read(band = 4, xoff = 0, yoff = row,
                            xsize = ncols, ysize = 1,
                            out_xsize = ncols, out_ysize = 1)
    row_b5 <- ds_b5vrt$read(band = 1, xoff = 0, yoff = row,
                            xsize = ncols, ysize = 1,
                            out_xsize = ncols, out_ysize = 1)
```

```
   rs$update(row_b5[row_fbfm == 165])
}
rs$get_count()
rs$get_mean()
rs$get_min()
rs$get_max()
rs$get_sum()
rs$get_var()
rs$get_sd()

ds_b5vrt$close()
ds_lcp$close()
ds_b5$close()
```

`read_ds` *Convenience wrapper for* `GDALRaster$read()`

Description

`read_ds()` will read from a raster dataset that is already open in a `GDALRaster` object. By default, it attempts to read the full raster extent from all bands at full resolution. `read_ds()` is sometimes more convenient than `GDALRaster$read()`, e.g., to read specific multiple bands for display with `plot_raster()`, or simply for the default arguments that read an entire raster into memory (see Note).

Usage

```
read_ds(
  ds,
  bands = NULL,
  xoff = 0,
  yoff = 0,
  xsize = ds$getRasterXSize(),
  ysize = ds$getRasterYSize(),
  out_xsize = xsize,
  out_ysize = ysize,
  as_list = FALSE,
  as_raw = FALSE
)
```

Arguments

`ds`	An object of class `GDALRaster` in open state.
`bands`	Integer vector of band numbers to read. By default all bands will be read.
`xoff`	Integer. The pixel (column) offset to the top left corner of the raster region to be read (zero to start from the left side).

yoff	Integer. The line (row) offset to the top left corner of the raster region to be read (zero to start from the top).
xsize	Integer. The width in pixels of the region to be read.
ysize	Integer. The height in pixels of the region to be read.
out_xsize	Integer. The width in pixels of the output buffer into which the desired region will be read (e.g., to read a reduced resolution overview).
out_ysize	Integer. The height in pixels of the output buffer into which the desired region will be read (e.g., to read a reduced resolution overview).
as_list	Logical. If TRUE, return output as a list of band vectors. If FALSE (the default), output is a vector of pixel data interleaved by band.
as_raw	Logical. If TRUE and the underlying data type is Byte, return output as R raw vector type. This maps to the setting $readByteAsRaw on the GDALRaster object, which will be temporarily updated in this function. To control this behavior in a persistent way on a dataset see $readByteAsRaw in GDALRaster-class.

Details

NA will be returned in place of the nodata value if the raster dataset has a nodata value defined for the band. Data are read as R integer type when possible for the raster data type (Byte, Int8, Int16, UInt16, Int32), otherwise as type double (UInt32, Float32, Float64).

The output object has attribute gis, a list containing:

```
$type = "raster"
$bbox = c(xmin, ymin, xmax, ymax)
$dim = c(xsize, ysize, nbands)
$srs = <projection as WKT2 string>
$datatype = <character vector of data type name by band>
```

The WKT version used for the projection string can be overridden by setting the OSR_WKT_FORMAT configuration option. See srs_to_wkt() for a list of supported values.

Value

If as_list = FALSE (the default), a vector of raw, integer, double or complex containing the values that were read. It is organized in left to right, top to bottom pixel order, interleaved by band. If as_list = TRUE, a list with number of elements equal to the number of bands read. Each element contains a vector of raw, integer, double or complex containing the pixel values that were read for the band.

Note

There is small overhead in calling read_ds() compared with calling GDALRaster$read() directly. This would only matter if calling the function repeatedly to read a raster in chunks. For the case of reading a large raster in many chunks, it will be optimal performance-wise to call GDALRaster$read() directly.

By default, this function will attempt to read the full raster into memory. It generally should not be called on large raster datasets using the default argument values. The memory size in bytes of

the returned vector will be, e.g., (xsize * ysize * number of bands * 4) for data read as `integer`, or (xsize * ysize * number of bands * 8) for data read as `double` (plus small object overhead for the vector).

See Also

`GDALRaster$read()`

Examples

```
# read three bands from a multi-band dataset
lcp_file <- system.file("extdata/storm_lake.lcp", package="gdalraster")
ds <- new(GDALRaster, lcp_file)

# as a vector of pixel data interleaved by band
r <- read_ds(ds, bands = c(6,5,4))
typeof(r)
length(r)
object.size(r)

# as a list of band vectors
r <- read_ds(ds, bands = c(6,5,4), as_list = TRUE)
typeof(r)
length(r)
object.size(r)

# gis attributes
attr(r, "gis")

ds$close()
```

`renameDataset` *Rename a dataset*

Description

`renameDataset()` renames a dataset in a format-specific way (e.g., rename associated files as appropriate). This could include moving the dataset to a new directory or even a new filesystem. The dataset should not be open in any existing `GDALRaster` objects when `renameDataset()` is called. Wrapper for `GDALRenameDataset()` in the GDAL API.

Usage

```
renameDataset(new_filename, old_filename, format = "")
```

Arguments

`new_filename`	New name for the dataset.
`old_filename`	Old name for the dataset (should not be open in a `GDALRaster` object).
`format`	Raster format short name (e.g., "GTiff"). If set to empty string `""` (the default), will attempt to guess the raster format from `old_filename`.

Value

Logical TRUE if no error or FALSE on failure.

Note

If format is set to an empty string "" (the default) then the function will try to identify the driver from old_filename. This is done internally in GDAL by invoking the Identify method of each registered GDALDriver in turn. The first driver that successful identifies the file name will be returned. An error is raised if a format cannot be determined from the passed file name.

See Also

GDALRaster-class, create(), createCopy(), deleteDataset(), copyDatasetFiles()

Examples

```
b5_file <- system.file("extdata/sr_b5_20200829.tif", package="gdalraster")
b5_tmp <- file.path(tempdir(), "b5_tmp.tif")
file.copy(b5_file,  b5_tmp)

ds <- new(GDALRaster, b5_tmp)
ds$buildOverviews("BILINEAR", levels = c(2, 4, 8), bands = c(1))
ds$getFileList()
ds$close()
b5_tmp2 <- file.path(tempdir(), "b5_tmp_renamed.tif")
renameDataset(b5_tmp2, b5_tmp)
ds <- new(GDALRaster, b5_tmp2)
ds$getFileList()
ds$close()

deleteDataset(b5_tmp2)
```

RunningStats-class *Class to calculate mean and variance in one pass*

Description

RunningStats computes summary statistics on a data stream efficiently. Mean and variance are calculated with Welford's online algorithm (https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance). The min, max, sum and count are also tracked. The input data values are not stored in memory, so this class can be used to compute statistics for very large data streams.

RunningStats is a C++ class exposed directly to R (via RCPP_EXPOSED_CLASS). Methods of the class are accessed using the $ operator.

Arguments

na_rm	Logical scalar. TRUE to remove NA from the input data (the default) or FALSE to retain NA.

Value

An object of class RunningStats. A RunningStats object maintains the current minimum, maximum, mean, variance, sum and count of values that have been read from the stream. It can be updated repeatedly with new values (i.e., chunks of data read from the input stream), but its memory footprint is negligible. Class methods for updating with new values, and retrieving the current values of statistics, are described in Details.

Usage (see Details)

```
## Constructor
rs <- new(RunningStats, na_rm)

## Methods
rs$update(newvalues)
rs$get_count()
rs$get_mean()
rs$get_min()
rs$get_max()
rs$get_sum()
rs$get_var()
rs$get_sd()
rs$reset()
```

Details

Constructor:

`new(RunningStats, na_rm)`
Returns an object of class RunningStats. The na_rm argument defaults to TRUE if omitted.

Methods:

`$update(newvalues)`
Updates the RunningStats object with a numeric vector of newvalues (i.e., a chunk of values from the data stream). No return value, called for side effects.

`$get_count()`
Returns the count of values received from the data stream.

`$get_mean()`
Returns the mean of values received from the data stream.

`$get_min()`
Returns the minimum value received from the data stream.

`$get_max()`
Returns the maximum value received from the data stream.

`$get_sum()`
Returns the sum of values received from the data stream.

`$get_var()`
Returns the variance of values from the data stream (denominator n - 1).

`$get_sd()`
Returns the standard deviation of values from the data stream (denominator n - 1).

`$reset()`
Clears the `RunningStats` object to its initialized state (count = 0). No return value, called for side effects.

Note

The intended use is computing summary statistics for specific subsets or zones of a raster that could be defined in various ways and are generally not contiguous. The algorithm as implemented here incurs the cost of floating point division for each new value updated (i.e., per pixel), but is reasonably efficient for the use case. Note that GDAL internally uses an optimized version of Welford's algorithm to compute raster statistics as described in detail by Rouault, 2016 (https://github.com/OSGeo/gdal/blob/master/gcore/statistics.txt). The class method `GDALRaster$getStatistics()` is a GDAL API wrapper that computes statistics for a whole raster band.

Examples

```
set.seed(42)
rs <- new(RunningStats, na_rm = TRUE)
rs

chunk <- runif(1000)
rs$update(chunk)
object.size(rs)

rs$get_count()
length(chunk)

rs$get_mean()
mean(chunk)

rs$get_min()
min(chunk)

rs$get_max()
max(chunk)

rs$get_var()
var(chunk)

rs$get_sd()
sd(chunk)


## 10^9 values read in 10,000 chunks
## should take under 1 minute on most PC hardware
for (i in 1:1e4) {
  chunk <- runif(1e5)
  rs$update(chunk)
}
rs$get_count()
rs$get_mean()
```

```
rs$get_var()

object.size(rs)
```

---

set_cache_max *Set the maximum memory size for the GDAL block cache*

Description

set_cache_max() sets the maximum amount of memory that GDAL is permitted to use for GDALRasterBlock caching. *The unit of the value to set is bytes.* Wrapper of GDALSetCacheMax64().

Usage

```
set_cache_max(nbytes)
```

Arguments

nbytes	A numeric value optionally carrying the integer64 class attribute (assumed to be a whole number, will be coerced to integer by truncation). Specifies the new cache size in bytes (maximum number of bytes for caching).

Value

No return value, called for side effects.

Note

This function will not make any attempt to check the consistency of the passed value with the effective capabilities of the OS.

It is recommended to consult the documentation for get_cache_max() and get_cache_used() before using this function.

get_cache_max(), get_cache_used()

Examples

```
(cachemax <- get_cache_max("bytes"))

set_cache_max(1e8)
get_cache_max()  # returns in MB by default

# reset to original
set_cache_max(cachemax)
get_cache_max()
```

set_config_option *Set GDAL configuration option*

Description

set_config_option() sets a GDAL runtime configuration option. Configuration options are essentially global variables the user can set. They are used to alter the default behavior of certain raster format drivers, and in some cases the GDAL core. For a full description and listing of available options see https://gdal.org/en/stable/user/configoptions.html.

Usage

```
set_config_option(key, value)
```

Arguments

key	Character name of a configuration option.
value	Character value to set for the option. value = "" (empty string) will unset a value previously set by set_config_option().

Value

No return value, called for side effects.

See Also

get_config_option()

vignette("gdal-config-quick-ref")

Examples

```
set_config_option("GDAL_CACHEMAX", "10%")
get_config_option("GDAL_CACHEMAX")
## unset:
set_config_option("GDAL_CACHEMAX", "")
```

sieveFilter *Remove small raster polygons*

Description

sieveFilter() is a wrapper for GDALSieveFilter() in the GDAL Algorithms API. It removes raster polygons smaller than a provided threshold size (in pixels) and replaces them with the pixel value of the largest neighbour polygon.

Usage

```
sieveFilter(
  src_filename,
  src_band,
  dst_filename,
  dst_band,
  size_threshold,
  connectedness,
  mask_filename = "",
  mask_band = 0L,
  options = NULL,
  quiet = FALSE
)
```

Arguments

src_filename	Filename of the source raster to be processed.
src_band	Band number in the source raster to be processed.
dst_filename	Filename of the output raster. It may be the same as src_filename to update the source file in place.
dst_band	Band number in dst_filename to write output. It may be the same as src_band to update the source raster in place.
size_threshold	Integer. Raster polygons with sizes (in pixels) smaller than this value will be merged into their largest neighbour.
connectedness	Integer. Either 4 indicating that diagonal pixels are not considered directly adjacent for polygon membership purposes, or 8 indicating they are.
mask_filename	Optional filename of raster to use as a mask.
mask_band	Band number in mask_filename to use as a mask. All pixels in the mask band with a value other than zero will be considered suitable for inclusion in polygons.
options	Algorithm options as a character vector of name=value pairs. None currently supported.
quiet	Logical scalar. If TRUE, a progress bar will not be displayed. Defaults to FALSE.

Details

Polygons are determined as regions of the raster where the pixels all have the same value, and that are contiguous (connected). Pixels determined to be "nodata" per the mask band will not be treated as part of a polygon regardless of their pixel values. Nodata areas will never be changed nor affect polygon sizes. Polygons smaller than the threshold with no neighbours that are as large as the threshold will not be altered. Polygons surrounded by nodata areas will therefore not be altered.

The algorithm makes three passes over the input file to enumerate the polygons and collect limited information about them. Memory use is proportional to the number of polygons (roughly 24 bytes per polygon), but is not directly related to the size of the raster. So very large raster files can be processed effectively if there aren't too many polygons. But extremely noisy rasters with many one pixel polygons will end up being expensive (in memory) to process.

The input dataset is read as integer data which means that floating point values are rounded to integers.

Value

Logical indicating success (invisible TRUE). An error is raised if the operation fails.

Examples

```
## remove single-pixel polygons from the vegetation type layer (EVT)
evt_file <- system.file("extdata/storml_evt.tif", package="gdalraster")

# create a blank raster to hold the output
evt_mmu_file <- file.path(tempdir(), "storml_evt_mmu2.tif")
rasterFromRaster(srcfile = evt_file,
                 dstfile = evt_mmu_file,
                 init = 32767)

# create a mask to exclude water pixels from the algorithm
# recode water (7292) to 0
expr <- "ifelse(EVT == 7292, 0, EVT)"
mask_file <- calc(expr = expr,
                  rasterfiles = evt_file,
                  var.names = "EVT")

# create a version of EVT with two-pixel minimum mapping unit
sieveFilter(src_filename = evt_file,
            src_band = 1,
            dst_filename = evt_mmu_file,
            dst_band = 1,
            size_threshold = 2,
            connectedness = 8,
            mask_filename = mask_file,
            mask_band = 1)
```

---

srs_convert *Convert spatial reference definitions to OGC WKT or PROJJSON*

Description

These functions convert various spatial reference formats to Well Known Text (WKT) or PROJJSON.

Usage

```
epsg_to_wkt(epsg, pretty = FALSE)

srs_to_wkt(srs, pretty = FALSE, gcs_only = FALSE)
```

```
srs_to_projjson(
  srs,
  multiline = TRUE,
  indent_width = 2L,
  schema = NA_character_
)
```

Arguments

epsg	Integer EPSG code.
pretty	Logical value. TRUE to return a nicely formatted WKT string for display to a person. FALSE for a regular WKT string (the default).
srs	Character string containing an SRS definition in various formats (see Details).
gcs_only	Logical value. TRUE to return only the definition of the GEOGCS node of the input srs. Defaults to FALSE (see Note).
multiline	Logical value. TRUE for PROJJSON multiline output (the default).
indent_width	Integer value. Defaults to 2. Only used if multiline = TRUE for PROJJSON output.
schema	Character string containing URL to PROJJSON schema. Can be set to empty string to disable it.

Details

epsg_to_wkt() exports the spatial reference for an EPSG code to WKT format. Wrapper for OSRImportFromEPSG() in the GDAL Spatial Reference System API with output to WKT.

srs_to_wkt() converts a spatial reference system (SRS) definition in various text formats to WKT. The function will examine the input SRS, try to deduce the format, and then export it to WKT. Wrapper for OSRSetFromUserInput() in the GDAL Spatial Reference System API with output to WKT.

srs_to_projjson() accepts a spatial reference system (SRS) definition in any of the formats supported by srs_to_wkt(), and converts into PROJJSON format. Wrapper for OSRExportToPROJJSON() in the GDAL Spatial Reference System API.

The input SRS may take the following forms:

- WKT - to convert WKT versions (see below)
- EPSG:n - EPSG code n
- AUTO:proj_id,unit_id,lon0,lat0 - WMS auto projections
- urn:ogc:def:crs:EPSG::n - OGC URNs
- PROJ.4 definitions
- filename - file to read for WKT, XML or PROJ.4 definition
- well known name such as NAD27, NAD83, WGS84 or WGS72
- IGNF:xxxx, ESRI:xxxx - definitions from the PROJ database
- PROJJSON (PROJ >= 6.2)

`srs_to_wkt()` is intended to be flexible, but by its nature it is imprecise as it must guess information about the format intended. `epsg_to_wkt()` could be used instead for EPSG codes.

As of GDAL 3.0, the default format for WKT export is OGC WKT 1. The WKT version can be overridden by using the OSR_WKT_FORMAT configuration option (see `set_config_option()`). Valid values are one of: SFSQL, WKT1_SIMPLE, WKT1, WKT1_GDAL, WKT1_ESRI, WKT2_2015, WKT2_2018, WKT2, DEFAULT. If SFSQL, a WKT1 string without AXIS, TOWGS84, AUTHORITY or EXTENSION node is returned. If WKT1_SIMPLE, a WKT1 string without AXIS, AUTHORITY or EXTENSION node is returned. WKT1 is an alias of WKT1_GDAL. WKT2 will default to the latest revision implemented (currently WKT2_2018). WKT2_2019 can be used as an alias of WKT2_2018 since GDAL 3.2

Value

Character string containing OGC WKT.

Note

Setting `gcs_only = TRUE` in `srs_to_wkt()` is a wrapper of `OSRCloneGeogCS()` in the GDAL API. The returned WKT will be for the GEOGCS node of the input `srs`.

See Also

srs_query

Examples

```
epsg_to_wkt(5070)
writeLines(epsg_to_wkt(5070, pretty = TRUE))

srs_to_wkt("NAD83")
writeLines(srs_to_wkt("NAD83", pretty = TRUE))
set_config_option("OSR_WKT_FORMAT", "WKT2")
writeLines(srs_to_wkt("NAD83", pretty = TRUE))
set_config_option("OSR_WKT_FORMAT", "")

srs_to_wkt("EPSG:5070", gcs_only = TRUE)

srs_to_projjson("NAD83") |> writeLines()
```

---

srs_query *Obtain information about a spatial reference system*

Description

Bindings to a subset of the GDAL Spatial Reference System API (https://gdal.org/en/stable/api/ogr_srs_api.html). These functions return various information about a spatial reference system passed as text in any of the formats supported by `srs_to_wkt()`.

Usage

```
srs_get_name(srs)

srs_find_epsg(srs, all_matches = FALSE)

srs_is_geographic(srs)

srs_is_derived_gcs(srs)

srs_is_local(srs)

srs_is_projected(srs)

srs_is_compound(srs)

srs_is_geocentric(srs)

srs_is_vertical(srs)

srs_is_dynamic(srs)

srs_is_same(
  srs,
  srs_other,
  criterion = "",
  ignore_axis_mapping = FALSE,
  ignore_coord_epoch = FALSE
)

srs_get_angular_units(srs)

srs_get_linear_units(srs)

srs_get_coord_epoch(srs)

srs_get_utm_zone(srs)

srs_get_axis_mapping_strategy(srs)
```

Arguments

srs	Character string containing an SRS definition in various formats (e.g., WKT, PROJ.4 string, well known name such as NAD27, NAD83, WGS84, etc., see srs_to_wkt()).
all_matches	Logical scalar. TRUE to return all identified matches in a data frame, including a confidence value (0-100) for each match. The default is FALSE which returns a character string in the form "EPSG:<code>" for the first match (highest confidence).

srs_other — Character string containing an SRS definition in various formats(see above).

criterion — Character string. One of STRICT, EQUIVALENT, EQUIVALENT_EXCEPT_AXIS_ORDER_GEOGCRS. Defaults to EQUIVALENT_EXCEPT_AXIS_ORDER_GEOGCRS.

ignore_axis_mapping — Logical scalar. If TRUE, sets IGNORE_DATA_AXIS_TO_SRS_AXIS_MAPPING=YES in the call to OSRIsSameEx() in the GDAL Spatial Reference System API. Defaults to NO.

ignore_coord_epoch — Logical scalar. If TRUE, sets IGNORE_COORDINATE_EPOCH=YES in the call to OSRIsSameEx() in the GDAL Spatial Reference System API. Defaults to NO.

Details

srs_find_epsg() tries to find a matching EPSG code. Matching may be partial, or may fail. If all_matches = TRUE, returns a data frame with entries sorted by decreasing match confidence (first entry has the highest match confidence); the default is FALSE which returns a character string in the form "EPSG:####" for the first match (highest confidence). Wrapper of OSRFindMatches() in the GDAL SRS API.

srs_get_name() returns the SRS name. Wrapper of OSRGetName() in the GDAL API.

srs_is_geographic() returns TRUE if the root is a GEOGCS node. Wrapper of OSRIsGeographic() in the GDAL API.

srs_is_derived_gcs() returns TRUE if the SRS is a derived geographic coordinate system (for example a rotated long/lat grid). Wrapper of OSRIsDerivedGeographic() in the GDAL API.

srs_is_local() returns TRUE if the SRS is a local coordinate system (the root is a LOCAL_CS node). Wrapper of OSRIsLocal() in the GDAL API.

srs_is_projected() returns TRUE if the SRS contains a PROJCS node indicating a it is a projected coordinate system. Wrapper of OSRIsProjected() in the GDAL API.

srs_is_compound() returns TRUE if the SRS is compound. Wrapper of OSRIsCompound() in the GDAL API.

srs_is_geocentric() returns TRUE if the SRS is a geocentric coordinate system. Wrapper of OSRIsGeocentric() in the GDAL API.

srs_is_vertical() returns TRUE if the SRS is a vertical coordinate system. Wrapper of OSRIsVertical() in the GDAL API.

srs_is_dynamic() returns TRUE if the SRS is is a dynamic coordinate system (relies on a dynamic datum, i.e., a datum that is not plate-fixed). Wrapper of OSRIsDynamic() in the GDAL API. Requires GDAL >= 3.4.

srs_is_same() returns TRUE if two spatial references describe the same system. Wrapper of OSRIsSame() in the GDAL API.

srs_get_angular_units() fetches the angular geographic coordinate system units. Returns a list of length two: the first element contains the unit name as a character string, and the second element contains a numeric value to multiply by angular distances to transform them to radians. Wrapper of OSRGetAngularUnits() in the GDAL API.

srs_get_linear_units() fetches the linear projection units. Returns a list of length two: the first element contains the unit name as a character string, and the second element contains a numeric

value to multiply by linear distances to transform them to meters. If no units are available, values of "Meters" and 1.0 will be assumed. Wrapper of `OSRGetLinearUnits()` in the GDAL API.

`srs_get_coord_epoch()` returns the coordinate epoch, as decimal year (e.g. 2021.3), or `0` if not set or not relevant. Wrapper of `OSRGetCoordinateEpoch()` in the GDAL API. Requires GDAL >= 3.4.

`srs_get_utm_zone()` returns the UTM zone number or zero if `srs` isn't a UTM definition. A positive value indicates northern hemisphere; a negative value is in the southern hemisphere. Wrapper of `OSRGetUTMZone()` in the GDAL API.

`srs_get_axis_mapping_strategy()` returns the data axis to CRS axis mapping strategy as a character string, one of:

- `OAMS_TRADITIONAL_GIS_ORDER`: for geographic CRS with lat/long order, the data will still be long/lat ordered. Similarly for a projected CRS with northing/easting order, the data will still be easting/northing ordered.
- `OAMS_AUTHORITY_COMPLIANT`: the data axis will be identical to the CRS axis.
- `OAMS_CUSTOM`: custom-defined data axis

See Also

srs_convert

Examples

```
wkt <- 'PROJCS["ETRS89 / UTM zone 32N (N-E)",
        GEOGCS["ETRS89",
            DATUM["European_Terrestrial_Reference_System_1989",
                SPHEROID["GRS 1980",6378137,298.257222101,
                    AUTHORITY["EPSG","7019"]],
                TOWGS84[0,0,0,0,0,0,0],
                AUTHORITY["EPSG","6258"]],
            PRIMEM["Greenwich",0,
                AUTHORITY["EPSG","8901"]],
            UNIT["degree",0.0174532925199433,
                AUTHORITY["EPSG","9122"]],
            AUTHORITY["EPSG","4258"]],
        PROJECTION["Transverse_Mercator"],
        PARAMETER["latitude_of_origin",0],
        PARAMETER["central_meridian",9],
        PARAMETER["scale_factor",0.9996],
        PARAMETER["false_easting",500000],
        PARAMETER["false_northing",0],
        UNIT["metre",1,
            AUTHORITY["EPSG","9001"]],
        AXIS["Northing",NORTH],
        AXIS["Easting",EAST]]'

srs_find_epsg(wkt)

srs_find_epsg(wkt, all_matches = TRUE)
```

```
srs_get_name("EPSG:5070")

srs_is_geographic("EPSG:5070")
srs_is_geographic("EPSG:4326")

srs_is_derived_gcs("WGS84")

srs_is_projected("EPSG:5070")
srs_is_projected("EPSG:4326")

srs_is_compound("EPSG:4326")

srs_is_geocentric("EPSG:7789")

srs_is_vertical("EPSG:5705")

f <- system.file("extdata/storml_elev.tif", package="gdalraster")
ds <- new(GDALRaster, f)

ds$getProjection() |> srs_is_projected()
ds$getProjection() |> srs_get_utm_zone()
ds$getProjection() |> srs_get_angular_units()
ds$getProjection() |> srs_get_linear_units()
ds$getProjection() |> srs_get_axis_mapping_strategy()

ds$getProjection() |> srs_is_same("EPSG:26912")
ds$getProjection() |> srs_is_same("NAD83")

ds$close()

# Requires GDAL >= 3.4
if (gdal_version_num() >= gdal_compute_version(3, 4, 0)) {
  if (srs_is_dynamic("WGS84"))
    print("WGS84 is dynamic")

  if (!srs_is_dynamic("NAD83"))
    print("NAD83 is not dynamic")
}
```

`transform_bounds` *Transform boundary*

Description

`transform_bounds()` transforms a bounding box, densifying the edges to account for nonlinear transformations along these edges and extracting the outermost bounds. Multiple bounding boxes may be given as rows of a numeric matrix or data frame. Wrapper of `OCTTransformBounds()` in the GDAL Spatial Reference System API. Requires GDAL >= 3.4.

Usage

```
transform_bounds(
  bbox,
  srs_from,
  srs_to,
  densify_pts = 21L,
  traditional_gis_order = TRUE
)
```

Arguments

bbox	Either a numeric vector of length four containing the input bounding box (xmin, ymin, xmax, ymax), or a four-column numeric matrix of bounding boxes (or data frame that can be coerced to a four-column numeric matrix).
srs_from	Character string specifying the spatial reference system for pts. May be in WKT format or any of the formats supported by srs_to_wkt().
srs_to	Character string specifying the output spatial reference system. May be in WKT format or any of the formats supported by srs_to_wkt().
densify_pts	Integer value giving the number of points to use to densify the bounding polygon in the transformation. Recommended to use 21 (the default).
traditional_gis_order	Logical value, TRUE to use traditional GIS order of axis mapping (the default) or FALSE to use authority compliant axis order (see Note).

Details

The following refer to the *output* values xmin, ymin, xmax, ymax:

If the destination CRS is geographic, the first axis is longitude, and xmax < xmin then the bounds crossed the antimeridian. In this scenario there are two polygons, one on each side of the antimeridian. The first polygon should be constructed with (xmin, ymin, 180, ymax) and the second with (-180, ymin, xmax, ymax).

If the destination CRS is geographic, the first axis is latitude, and ymax < ymin then the bounds crossed the antimeridian. In this scenario there are two polygons, one on each side of the antimeridian. The first polygon should be constructed with (ymin, xmin, ymax, 180) and the second with (ymin, -180, ymax, xmax).

Value

For a single input bounding box, a numeric vector of length four containing the transformed bounding box in the output spatial reference system (xmin, ymin, xmax, ymax). For input of multiple bounding boxes, a four-column numeric matrix with each row containing the corresponding transformed bounding box (xmin, ymin, xmax, ymax).

Note

traditional_gis_order = TRUE (the default) means that for geographic CRS with lat/long order, the data will still be long/lat ordered. Similarly for a projected CRS with northing/easting order, the data will still be easting/northing ordered (GDAL's OAMS_TRADITIONAL_GIS_ORDER).

traditional_gis_order = FALSE means that the data axis will be identical to the CRS axis (GDAL's OAMS_AUTHORITY_COMPLIANT).

See https://gdal.org/en/stable/tutorials/osr_api_tut.html#crs-and-axis-order.

Examples

```
bb <- c(-1405880.71737, -1371213.76254, 5405880.71737, 5371213.76254)

# traditional GIS axis ordering by  default (lon, lat)
transform_bounds(bb, "EPSG:32761", "EPSG:4326")

# authority compliant axis ordering
transform_bounds(bb, "EPSG:32761", "EPSG:4326",
                 traditional_gis_order = FALSE)
```

---

transform_xy *Transform geospatial x/y coordinates*

Description

transform_xy() transforms geospatial x, y coordinates to a new projection. The input points may optionally have z vertices (x, y, z) or time values (x, y, z, t). Wrapper for OGRCoordinateTransformation::Transform() in the GDAL Spatial Reference System C++ API.

Usage

```
transform_xy(pts, srs_from, srs_to)
```

Arguments

pts	A data frame or numeric matrix containing geospatial point coordinates, or point geometries as a list of WKB raw vectors or character vector of WKT strings. If data frame or matrix, the number of columns must be either two (x, y), three (x, y, z) or four (x, y, z, t). May be also be given as a numeric vector for one point (xy, xyz, or xyzt).
srs_from	Character string specifying the spatial reference system for pts. May be in WKT format or any of the formats supported by srs_to_wkt().
srs_to	Character string specifying the output spatial reference system. May be in WKT format or any of the formats supported by srs_to_wkt().

Value

Numeric matrix of geospatial (x, y) coordinates in the projection specified by srs_to (potentially also with z, or z and t columns).

Note

transform_xy() uses traditional GIS order for the input and output xy (i.e., longitude/latitude ordered for geographic coordinates).

Input points that contain missing values (NA) will be assigned NA in the output and a warning emitted. Input points that fail to transform with the GDAL API call will also be assigned NA in the output with a specific warning indicating that case.

See Also

srs_to_wkt(), inv_project()

Examples

```
pt_file <- system.file("extdata/storml_pts.csv", package="gdalraster")
pts <- read.csv(pt_file)
print(pts)
# id, x, y in NAD83 / UTM zone 12N
# transform to NAD83 / CONUS Albers
transform_xy(pts = pts[, -1], srs_from = "EPSG:26912", srs_to = "EPSG:5070")
```

---

translate *Convert raster data between different formats*

Description

translate() is a wrapper of the gdal_translate command-line utility (see https://gdal.org/en/stable/programs/gdal_translate.html). The function can be used to convert raster data between different formats, potentially performing some operations like subsetting, resampling, and rescaling pixels in the process. Refer to the GDAL documentation at the URL above for a list of command-line arguments that can be passed in cl_arg.

Usage

```
translate(src_filename, dst_filename, cl_arg = NULL, quiet = FALSE)
```

Arguments

src_filename	Either a character string giving the filename of the source raster, or an object of class GDALRaster for the source.
dst_filename	Character string. Filename of the output raster.
cl_arg	Optional character vector of command-line arguments for gdal_translate (see URL above).
quiet	Logical scalar. If TRUE, a progress bar will not be displayed. Defaults to FALSE.

Value

Logical indicating success (invisible TRUE). An error is raised if the operation fails.

See Also

GDALRaster-class, rasterFromRaster(), warp()

ogr2ogr() for vector data

Examples

```
# convert the elevation raster to Erdas Imagine format and resample to 90m
elev_file <- system.file("extdata/storml_elev.tif", package="gdalraster")
img_file <- file.path(tempdir(), "storml_elev_90m.img")

# command-line arguments for gdal_translate
args <- c("-tr", "90", "90", "-r", "average")
args <- c(args, "-of", "HFA", "-co", "COMPRESSED=YES")

translate(elev_file, img_file, args)

ds <- new(GDALRaster, img_file)
ds$info()

ds$close()
```

---

validateCreationOptions

Validate the list of creation options that are handled by a driver

---

Description

validateCreationOptions() is a helper function primarily used by GDAL's Create() and CreateCopy() to validate that the passed-in list of creation options is compatible with the GDAL_DMD_CREATIONOPTIONLIST metadata item defined by some drivers. If the GDAL_DMD_CREATIONOPTIONLIST metadata item is not defined, this function will return TRUE. Otherwise it will check that the keys and values in the list of creation options are compatible with the capabilities declared by the GDAL_DMD_CREATIONOPTIONLIST metadata item. In case of incompatibility a message will be emitted and FALSE will be returned. Wrapper of GDALValidateCreationOptions() in the GDAL API.

Usage

```
validateCreationOptions(format, options)
```

Arguments

format	Character string giving a format driver short name (e.g., "GTiff").
options	A character vector of format-specific creation options as "NAME=VALUE" pairs.

Value

A logical value, TRUE if the given creation options are compatible with the capabilities declared by the GDAL_DMD_CREATIONOPTIONLIST metadata item for the specified format driver (or if the GDAL_DMD_CREATIONOPTIONLIST metadata item is not defined for this driver), otherwise FALSE.

See Also

getCreationOptions(), create(), createCopy()

Examples

```
validateCreationOptions("GTiff", c("COMPRESS=LZW", "TILED=YES"))
```

VSIFile-class *Class wrapping the GDAL VSIVirtualHandle API for binary file I/O*

Description

VSIFile provides bindings to the GDAL VSIVirtualHandle API. Encapsulates a VSIVirtualHandle (https://gdal.org/en/stable/api/cpl_cpp.html#_CPPv416VSIVirtualHandle). This API abstracts binary file I/O across "regular" file systems, URLs, cloud storage services, Zip/GZip/7z/RAR, and in-memory files. It provides analogs of several Standard C file I/O functions, allowing virtualization of disk I/O so that non-file data sources can be made to appear as files.

VSIFile is a C++ class exposed directly to R (via RCPP_EXPOSED_CLASS). Methods of the class are accessed using the $ operator.

Arguments

filename	Character string containing the filename to open. It may be a file in a regular local filesystem, or a filename with a GDAL /vsiPREFIX/ (see https://gdal.org/en/stable/user/virtual_file_systems.html).
access	Character string containing the access requested (i.e., "r", "r+", "w", "w+). Defaults to "r". Binary access is always implied and the "b" does not need to be included in access.

Access	Explanation	If file exists
"r"	open file for reading	read from start
"r+"	open file for read/write	read from start
"w"	create file for writing	destroy contents
"w+"	create file for read/write	destroy contents

options	Optional character vector of NAME=VALUE pairs specifying filesystem-dependent options (GDAL >= 3.3, see Details).

Value

An object of class `VSIFile` which contains a pointer to a `VSIVirtualHandle`, and methods that operate on the file as described in Details.

Usage (see Details)

```
## Constructors
vf <- new(VSIFile, filename)
# specifying access:
vf <- new(VSIFile, filename, access)
# specifying access and options (both required):
vf <- new(VSIFile, filename, access, options)

## Methods
vf$seek(offset, origin)
vf$tell()
vf$rewind()
vf$read(nbytes)
vf$write(object)
vf$eof()
vf$truncate(new_size)
vf$flush()
vf$ingest(max_size)

vf$close()
vf$open()
vf$get_filename()
vf$get_access()
vf$set_access(access)
```

Details

Constructors:

`new(VSIFile, filename)`
Returns an object of class `VSIFile`, or an error is raised if a file handle cannot be obtained.

`new(VSIFile, filename, access)`
Alternate constructor for passing `access` as a character string (e.g., `"r"`, `"r+"`, `"w"`, `"w+"`). Returns an object of class `VSIFile` with an open file handle, or an error is raised if a file handle cannot be obtained.

`new(VSIFile, filename, access, options)`
Alternate constructor for passing `access` as a character string, and `options` as a character vector of "NAME=VALUE" pairs (all arguments required, GDAL >= 3.3 required for `options` support).
The `options` argument is highly file system dependent. Supported options as of GDAL 3.9 include:

- MIME headers such as Content-Type and Content-Encoding are supported for the /vsis3/, /vsigs/, /vsiaz/, /vsiadls/ file systems.
- DISABLE_READDIR_ON_OPEN=YES/NO (GDAL >= 3.6) for /vsicurl/ and other network-based file systems. By default, directory file listing is done, unless YES is specified.

- WRITE_THROUGH=YES (GDAL >= 3.8) for Windows regular files to set the FILE_FLAG_WRITE_THROUGH flag to the `CreateFile()` function. In that mode, the data are written to the system cache but are flushed to disk without delay.

Methods:

`$seek(offset, origin)`
Seek to a requested `offset` in the file. `offset` is given as a positive numeric scalar, optionally as `bit64::integer64` type. `origin` is given as a character string, one of `SEEK_SET`, `SEEK_CUR` or `SEEK_END`. Package global constants are defined for convenience, so these can be passed unquoted. Note that `offset` is an unsigned type, so `SEEK_CUR` can only be used for positive seek. If negative seek is needed, use:

```
vf$seek(vf$tell() + negative_offset, SEEK_SET)
```

Returns `0` on success or `-1` on failure.

`$tell()`
Returns the current file read/write offset in bytes from the beginning of the file. The return value is a numeric scalar carrying the `integer64` class attribute.

`$rewind()`
Rewind the file pointer to the beginning of the file. This is equivalent to `vf$seek(0, SEEK_SET)`. No return value, called for that side effect.

`$read(nbytes)`
Read `nbytes` bytes from the file at the current offset. Returns an R `raw` vector, or `NULL` if the operation fails.

`$write(object)`
Write bytes to the file at the current offset. `object` is a `raw` vector. Returns the number of bytes successfully written, as numeric scalar carrying the `integer64` class attribute. See also base R `charToRaw()` / `rawToChar()`, convert to or from raw vectors, and `readBin()` / `writeBin()` which read binary data from or write binary data to a raw vector.

`$eof()`
Test for end of file. Returns `TRUE` if an end-of-file condition occurred during the previous read operation. The end-of-file flag is cleared by a successful call to `$seek()`.

`$truncate(new_size)`
Truncate/expand the file to the specified `new_size`, given as a positive numeric scalar, optionally as `bit64::integer64` type. Returns `0` on success.

`$flush()`
Flush pending writes to disk. For files in write or update mode and on file system types where it is applicable, all pending output on the file is flushed to the physical disk. On Windows regular files, this method does nothing, unless the `VSI_FLUSH=YES` configuration option is set (and only when the file has not been opened with the `WRITE_THROUGH` option). Returns `0` on success or `-1` on error.

`$ingest(max_size)`
Ingest a file into memory. Read the whole content of the file into a `raw` vector. `max_size` is the maximum size of file allowed, given as a numeric scalar, optionally as `bit64::integer64` type. If no limit, set to a negative value. Returns a `raw` vector, or `NULL` if the operation fails.

`$close()`
Closes the file. The file should always be closed when I/O has been completed. Returns `0` on success or `-1` on error.

`$open()`
This method can be used to re-open the file after it has been closed, using the same `filename`, and same `options` if any are set. The file will be opened using `access` as currently set. The `$set_access()` method can be called to change the requested access while the file is closed. No return value. An error is raised if a file handle cannot be obtained.

`$get_filename()`
Returns a character string containing the `filename` associated with this `VSIFile` object (the `filename` originally used to create the object).

`$get_access()`
Returns a character string containing the `access` as currently set on this `VSIFile` object.

`$set_access(access)`
Sets the requested read/write access on this `VSIFile` object, given as a character string (i.e., `"r"`, `"r+"`, `"w"`, `"w+"`). The access can be changed only while the `VSIFile` object is closed, and will apply when it is re-opened with a call to `$open()`. Returns `0` on success or `-1` on error.

Note

File offsets are given as R `numeric` (i.e., `double` type), optionally carrying the `bit64::integer64` class attribute. They are returned as `numeric` with the `integer64` class attribute attached. The `integer64` type is signed, so the maximum file offset supported by this interface is `9223372036854775807` (the value of `bit64::lim.integer64()[2]`).

Some virtual file systems allow only sequential write, so no seeks or read operations are then allowed (e.g., AWS S3 files with /vsis3/). Starting with GDAL 3.2, a configuration option can be set with:

```
set_config_option("CPL_VSIL_USE_TEMP_FILE_FOR_RANDOM_WRITE", "YES")
```

in which case random-write access is possible (involves the creation of a temporary local file, whose location is controlled by the `CPL_TMPDIR` configuration option). In this case, setting `access` to `"w+"` may be needed for writing with seek and read operations (if creating a new file, otherwise, `"r+"` to open an existing file), while `"w"` access would allow sequential write only.

See Also

GDAL Virtual File Systems (compressed, network hosted, etc...):
/vsimem, /vsizip, /vsitar, /vsicurl, ...
https://gdal.org/en/stable/user/virtual_file_systems.html

`vsi_copy_file()`, `vsi_read_dir()`, `vsi_stat()`, `vsi_unlink()`

Examples

```
# The examples make use of the FARSITE LCP format specification at:
# https://gdal.org/en/stable/drivers/raster/lcp.html
# An LCP file is a raw format with a 7,316-byte header. The format
# specification gives byte offets and data types for fields in the header.

lcp_file <- system.file("extdata/storm_lake.lcp", package="gdalraster")

# identify a FARSITE v.4 LCP file
```

```
# function to check if the first three fields have valid data
# input is the first twelve raw bytes in the file
is_lcp <- function(bytes) {
  values <- readBin(bytes, "integer", n = 3)
  if ((values[1] == 20 || values[1] == 21) &&
      (values[2] == 20 || values[2] == 21) &&
      (values[3] >= -90 && values[3] <= 90)) {

    return(TRUE)
  } else {
    return(FALSE)
  }
}

vf <- new(VSIFile, lcp_file)
vf

vf$read(12) |> is_lcp()

vf$tell()

# read the whole file into memory
bytes <- vf$ingest(-1)
vf$close()

# write to a VSI in-memory file
mem_file <- "/vsimem/storml_copy.lcp"
vf <- new(VSIFile, mem_file, "w+")
vf$write(bytes)

vf$tell()
vf$rewind()
vf$tell()

vf$seek(0, SEEK_END)
(vf$tell() == vsi_stat(lcp_file, "size"))  # TRUE

vf$rewind()
vf$read(12) |> is_lcp()

# read/write an integer field
# write invalid data for the Latitude field and then set back
# save the original first
vf$seek(8, SEEK_SET)
lat_orig <- vf$read(4)
readBin(lat_orig, "integer")  # 46
# latitude -99 out of range
vf$seek(8, SEEK_SET)
writeBin(-99L, raw()) |> vf$write()
vf$rewind()
vf$read(12) |> is_lcp()  # FALSE
vf$seek(8, SEEK_SET)
vf$read(4) |> readBin("integer")  # -99
```

```
# set back to original
vf$seek(8, SEEK_SET)
vf$write(lat_orig)
vf$rewind()
vf$read(12) |> is_lcp()  # TRUE

# read a vector of doubles - xmax, xmin, ymax, ymin
# 327766.1, 323476.1, 5105082.0, 5101872.0
vf$seek(4172, SEEK_SET)
vf$read(32) |> readBin("double", n = 4)

# read a short int, the canopy cover units
vf$seek(4232, SEEK_SET)
vf$read(2) |> readBin("integer", size = 2)  # 1 = "percent"

# read the Description field
vf$seek(6804, SEEK_SET)
bytes <- vf$read(512)
rawToChar(bytes)

# edit the Description
desc <- paste(rawToChar(bytes),
              "Storm Lake AOI,",
              "Beaverhead-Deerlodge National Forest, Montana.")

vf$seek(6804, SEEK_SET)
charToRaw(desc) |> vf$write()
vf$close()

# verify the file as a raster dataset
ds <- new(GDALRaster, mem_file)
ds$info()

# retrieve Description from the metadata
# band = 0 for dataset-level metadata, domain = "" for default domain
ds$getMetadata(band = 0, domain = "")
ds$getMetadataItem(band = 0, mdi_name = "DESCRIPTION", domain = "")

ds$close()
```

---

vsi_clear_path_options

Clear path specific configuration options

Description

`vsi_clear_path_options()` clears path specific options previously set with `vsi_set_path_option()`. Wrapper for `VSIClearPathSpecificOptions()` in the GDAL Common Portability Library. Requires GDAL >= 3.6.

Usage

```
vsi_clear_path_options(path_prefix)
```

Arguments

path_prefix	Character string. If set to "" (empty string), all path specific options are cleared. If set to a path prefix, only those options set with vsi_set_path_option(path_prefix, ...) will be cleared.

Value

No return value, called for side effect.

Note

No particular care is taken to remove options from RAM in a secure way.

See Also

vsi_set_path_option()

---

vsi_constants *Constants for VSIFile$seek()*

Description

These are package global constants for convenience in calling VSIFile$seek().

Usage

```
SEEK_SET

SEEK_CUR

SEEK_END
```

Format

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

vsi_copy_file *Copy a source file to a target filename*

Description

vsi_copy_file() is a wrapper for VSICopyFile() in the GDAL Common Portability Library. The GDAL VSI functions allow virtualization of disk I/O so that non file data sources can be made to appear as files. See https://gdal.org/en/stable/user/virtual_file_systems.html. Requires GDAL >= 3.7.

Usage

```
vsi_copy_file(src_file, target_file, show_progress = FALSE)
```

Arguments

src_file	Character string. Filename of the source file.
target_file	Character string. Filename of the target file.
show_progress	Logical scalar. If TRUE, a progress bar will be displayed (the size of src_file will be retrieved in GDAL with VSIStatL()). Default is FALSE.

Details

The following copies are made fully on the target server, without local download from source and upload to target:

- /vsis3/ -> /vsis3/
- /vsigs/ -> /vsigs/
- /vsiaz/ -> /vsiaz/
- /vsiadls/ -> /vsiadls/
- any of the above or /vsicurl/ -> /vsiaz/ (starting with GDAL 3.8)

Value

0 on success or -1 on an error.

Note

If target_file has the form /vsizip/foo.zip/bar, the default options described for the function addFilesInZip() will be in effect.

See Also

copyDatasetFiles(), vsi_stat(), vsi_sync()

Examples

```
elev_file <- system.file("extdata/storml_elev.tif", package="gdalraster")
tmp_file <- "/vsimem/elev_temp.tif"

# Requires GDAL >= 3.7
if (gdal_version_num() >= gdal_compute_version(3, 7, 0)) {
  result <- vsi_copy_file(elev_file, tmp_file)
  (result == 0)
  print(vsi_stat(tmp_file, "size"))

  vsi_unlink(tmp_file)
}
```

---

vsi_curl_clear_cache *Clean cache associated with /vsicurl/ and related file systems*

Description

vsi_curl_clear_cache() cleans the local cache associated with /vsicurl/ (and related file systems). This function is a wrapper for VSICurlClearCache() and VSICurlPartialClearCache() in the GDAL Common Portability Library. See Details for the GDAL documentation.

Usage

```
vsi_curl_clear_cache(partial = FALSE, file_prefix = "", quiet = TRUE)
```

Arguments

partial	Logical scalar. Whether to clear the cache only for a given filename (see Details).
file_prefix	Character string. Filename prefix to use if partial = TRUE.
quiet	Logical scalar. TRUE (the default) to wrap the API call in a quiet error handler, or FALSE to print any potential error messages to the console.

Details

/vsicurl/ (and related file systems like /vsis3/, /vsigs/, /vsiaz/, /vsioss/, /vsiswift/) cache a number of metadata and data for faster execution in read-only scenarios. But when the content on the server-side may change during the same process, those mechanisms can prevent opening new files, or give an outdated version of them. If partial = TRUE, cleans the local cache associated for a given filename (and its subfiles and subdirectories if it is a directory).

Value

No return value, called for side effects.

Examples

```
vsi_curl_clear_cache()
```

vsi_get_actual_url *Returns the actual URL of a supplied VSI filename*

Description

vsi_get_actual_url() returns the actual URL of a supplied filename. Currently only returns a non-NULL value for network-based virtual file systems. For example "/vsis3/bucket/filename" will be expanded as "https://bucket.s3.amazon.com/filename". Wrapper for VSIGetActualURL() in the GDAL API.

Usage

```
vsi_get_actual_url(filename)
```

Arguments

filename	Character string containing a /vsiPREFIX/ filename.

Value

Character string containing the actual URL, or NULL if filename is not a network-based virtual file system.

See Also

vsi_get_signed_url()

Examples

```
## Not run:
f <- "/vsiaz/items/io-lulc-9-class.parquet"
set_config_option("AZURE_STORAGE_ACCOUNT", "pcstacitems")
# token obtained from:
# https://planetarycomputer.microsoft.com/api/sas/v1/token/pcstacitems/items
set_config_option("AZURE_STORAGE_SAS_TOKEN","<token>")
vsi_get_actual_url(f)
#> [1] "https://pcstacitems.blob.core.windows.net/items/io-lulc-9-class.parquet"
vsi_get_signed_url(f)
#> [1] "https://pcstacitems.blob.core.windows.net/items/io-lulc-9-class.parquet?<token>"

## End(Not run)
```

vsi_get_disk_free_space *Return free disk space available on the filesystem*

Description

vsi_get_disk_free_space() returns the free disk space available on the filesystem. Wrapper for VSIGetDiskFreeSpace() in the GDAL Common Portability Library.

Usage

```
vsi_get_disk_free_space(path)
```

Arguments

path	Character string. A directory of the filesystem to query.

Value

Numeric scalar. The free space in bytes (as bit64::integer64 type), or -1 in case of error.

Examples

```
tmp_dir <- file.path("/vsimem", "tmpdir")
vsi_mkdir(tmp_dir)
vsi_get_disk_free_space(tmp_dir)
vsi_rmdir(tmp_dir)
```

vsi_get_file_metadata *Get metadata on files*

Description

vsi_get_file_metadata() returns metadata for file system objects. Implemented for network-like filesystems. Starting with GDAL 3.7, implemented for /vsizip/ with SOZip metadata. Wrapper of VSIGetFileMetadata() in the GDAL Common Portability Library.

Usage

```
vsi_get_file_metadata(filename, domain)
```

Arguments

filename	Character string. The path of the file system object to be queried.
domain	Character string. Metadata domain to query. Depends on the file system, see Details.

Details

The metadata available depends on the file system. The following are supported as of GDAL 3.9:

- HEADERS: to get HTTP headers for network-like filesystems (/vsicurl/, /vsis3/, /vsgis/, etc).
- TAGS: for /vsis3/, to get S3 Object tagging information. For /vsiaz/, to get blob tags.
- STATUS: specific to /vsiadls/: returns all system-defined properties for a path (seems in practice to be a subset of HEADERS).
- ACL: specific to /vsiadls/ and /vsigs/: returns the access control list for a path. For /vsigs/, a single `XML=xml_content` string is returned.
- METADATA: specific to /vsiaz/: blob metadata (this will be a subset of what `domain=HEADERS` returns).
- ZIP: specific to /vsizip/: to obtain ZIP specific metadata, in particular if a file is SOZIP-enabled (`SOZIP_VALID=YES`).

Value

A named list of values, or `NULL` in case of error or empty list.

See Also

`vsi_stat()`, `addFilesInZip()`

Examples

```
# validate an SOZip-enabled file
# Requires GDAL >= 3.7
f <- system.file("extdata/ynp_features.zip", package = "gdalraster")

zf <- file.path("/vsizip", f)
# files in zip archive
vsi_read_dir(zf)

# SOZip metadata for ynp_features.gpkg
zf_gpkg <- file.path(zf, "ynp_features.gpkg")
vsi_get_file_metadata(zf_gpkg, domain = "ZIP")
```

`vsi_get_fs_options` *Return the list of options associated with a virtual file system handler*

Description

`vsi_get_fs_options()` returns the list of options associated with a virtual file system handler. Those options may be set as configuration options with `set_config_option()`. Wrapper for `VSIGetFileSystemOptions()` in the GDAL API.

Usage

```
vsi_get_fs_options(filename, as_list = TRUE)
```

Arguments

filename	Filename, or prefix of a virtual file system handler.
as_list	Logical scalar. If TRUE (the default), the XML string returned by GDAL will be coerced to list. FALSE to return the configuration options as a serialized XML string.

Value

An XML string, or empty string ("") if no options are declared. If as_list = TRUE (the default), the XML string will be coerced to list with xml2::as_list().

See Also

set_config_option(), vsi_get_fs_prefixes()

https://gdal.org/en/stable/user/virtual_file_systems.html

Examples

```
vsi_get_fs_options("/vsimem/")

vsi_get_fs_options("/vsizip/")

vsi_get_fs_options("/vsizip/", as_list = FALSE)
```

---

vsi_get_fs_prefixes *Return the list of virtual file system handlers currently registered*

Description

vsi_get_fs_prefixes() returns the list of prefixes for virtual file system handlers currently registered (e.g., "/vsimem/", "/vsicurl/", etc). Wrapper for VSIGetFileSystemsPrefixes() in the GDAL API.

Usage

```
vsi_get_fs_prefixes()
```

Value

Character vector containing prefixes of the virtual file system handlers.

See Also

vsi_get_fs_options()

https://gdal.org/en/stable/user/virtual_file_systems.html

Examples

```
vsi_get_fs_prefixes()
```

---

vsi_get_signed_url *Returns a signed URL for a supplied VSI filename*

Description

vsi_get_signed_url() Returns a signed URL of a supplied filename. Currently only returns a non-NULL value for /vsis3/, /vsigs/, /vsiaz/ and /vsioss/ For example "/vsis3/bucket/filename" will be expanded as "https://bucket.s3.amazon.com/filename?X-Amz-Algorithm=AWS4-HMAC-SHA256..." Configuration options that apply for file opening (typically to provide credentials), and are returned by vsi_get_fs_options(), are also valid in that context. Wrapper for VSIGetSignedURL() in the GDAL API.

Usage

```
vsi_get_signed_url(filename, options = NULL)
```

Arguments

filename	Character string containing a /vsiPREFIX/ filename.
options	Character vector of NAME=VALUE pairs (see Details).

Details

The options argument accepts a character vector of name=value pairs. For /vsis3/, /vsigs/, /vsiaz/ and /vsioss/, the following options are supported:

- START_DATE=YYMMDDTHHMMSSZ: date and time in UTC following ISO 8601 standard, corresponding to the start of validity of the URL. If not specified, current date time.
- EXPIRATION_DELAY=number_of_seconds: number between 1 and 604800 (seven days) for the validity of the signed URL. Defaults to 3600 (one hour).
- VERB=GET/HEAD/DELETE/PUT/POST: HTTP VERB for which the request will be used. Defaults to GET.

/vsiaz/ supports additional options:

- SIGNEDIDENTIFIER=value: to relate the given shared access signature to a corresponding stored access policy.
- SIGNEDPERMISSIONS=r|w: permissions associated with the shared access signature. Normally deduced from VERB.

Value

Character string containing the signed URL, or NULL if filename is not a network-based virtual file system.

See Also

vsi_get_actual_url()

Examples

```
## Not run:
f <- "/vsiaz/items/io-lulc-9-class.parquet"
set_config_option("AZURE_STORAGE_ACCOUNT", "pcstacitems")
# token obtained from:
# https://planetarycomputer.microsoft.com/api/sas/v1/token/pcstacitems/items
set_config_option("AZURE_STORAGE_SAS_TOKEN", "<token>")
vsi_get_actual_url(f)
#> [1] "https://pcstacitems.blob.core.windows.net/items/io-lulc-9-class.parquet"
vsi_get_signed_url(f)
#> [1] "https://pcstacitems.blob.core.windows.net/items/io-lulc-9-class.parquet?<token>"

## End(Not run)
```

---

vsi_is_local *Returns if the file/filesystem is "local".*

Description

vsi_is_local() returns whether the file/filesystem is "local". Wrapper for VSIIsLocal() in the GDAL API. Requires GDAL >= 3.6.

Usage

```
vsi_is_local(filename)
```

Arguments

filename	Character string. The path of the filesystem object to be tested.

Value

Logical scalar. TRUE if if the input file path is local.

Note

The concept of local is mostly by opposition with a network / remote file system whose access time can be long.

/vsimem/ is considered to be a local file system, although a non-persistent one.

Examples

```
# Requires GDAL >= 3.6
if (gdal_version_num() >= gdal_compute_version(3, 6, 0))
  print(vsi_is_local("/vsimem/test-mem-file.tif"))
```

---

vsi_mkdir *Create a directory*

Description

`vsi_mkdir()` creates a new directory with the indicated mode. For POSIX-style systems, the mode is modified by the file creation mask (umask). However, some file systems and platforms may not use umask, or they may ignore the mode completely. So a reasonable cross-platform default mode value is `0755`. With `recursive = TRUE`, creates a directory and all its ancestors. This function is a wrapper for `VSIMkdir()` and `VSIMkdirRecursive()` in the GDAL Common Portability Library.

Usage

```
vsi_mkdir(path, mode = "0755", recursive = FALSE)
```

Arguments

`path`	Character string. The path to the directory to create.
`mode`	Character string. The permissions mode in octal with prefix `0`, e.g., `"0755"` (the default).
`recursive`	Logical scalar. `TRUE` to create the directory and its ancestors. Defaults to `FALSE`.

Value

`0` on success or `-1` on an error.

See Also

`vsi_read_dir()`, `vsi_rmdir()`

Examples

```
new_dir <- file.path(tempdir(), "newdir")
vsi_mkdir(new_dir)
vsi_stat(new_dir, "type")
vsi_rmdir(new_dir)
```

vsi_read_dir *Read names in a directory*

Description

vsi_read_dir() abstracts access to directory contents. It returns a character vector containing the names of files and directories in this directory. With recursive = TRUE, reads the list of entries in the directory and subdirectories. This function is a wrapper for VSIReadDirEx() and VSIReadDirRecursive() in the GDAL Common Portability Library.

Usage

```
vsi_read_dir(path, max_files = 0L, recursive = FALSE, all_files = FALSE)
```

Arguments

path	Character string. The relative or absolute path of a directory to read.
max_files	Integer scalar. The maximum number of files after which to stop, or 0 for no limit (see Note). Ignored if recursive = TRUE.
recursive	Logical scalar. TRUE to read the directory and its subdirectories. Defaults to FALSE.
all_files	Logical scalar. If FALSE (the default), only the names of visible files are returned (following Unix-style visibility, that is files whose name does not start with a dot). If TRUE, all file names will be returned.

Value

A character vector containing the names of files and directories in the directory given by path. The listing is in alphabetical order, and does not include the special entries '.' and '..' even if they are present in the directory. An empty string ("") is returned if path does not exist.

Note

If max_files is set to a positive number, directory listing will stop after that limit has been reached. Note that to indicate truncation, at least one element more than the max_files limit will be returned. If the length of the returned character vector is lesser or equal to max_files, then no truncation occurred. The max_files parameter is ignored when recursive = TRUE.

See Also

vsi_mkdir(), vsi_rmdir(), vsi_stat(), vsi_sync()

Examples

```
# regular file system for illustration
data_dir <- system.file("extdata", package="gdalraster")
vsi_read_dir(data_dir)
```

vsi_rename *Rename a file*

Description

vsi_rename() renames a file object in the file system. The GDAL documentation states it should be possible to rename a file onto a new filesystem, but it is safest if this function is only used to rename files that remain in the same directory. This function goes through the GDAL VSIFileHandler virtualization and may work on unusual filesystems such as in memory. It is a wrapper for VSIRename() in the GDAL Common Portability Library. Analog of the POSIX rename() function.

Usage

```
vsi_rename(oldpath, newpath)
```

Arguments

oldpath	Character string. The name of the file to be renamed.
newpath	Character string. The name the file should be given.

Value

0 on success or -1 on an error.

See Also

renameDataset(), vsi_copy_file()

Examples

```
# regular file system for illustration
elev_file <- system.file("extdata/storml_elev.tif", package="gdalraster")
tmp_file <- tempfile(fileext = ".tif")
file.copy(elev_file, tmp_file)
new_file <- file.path(dirname(tmp_file), "storml_elev_copy.tif")
vsi_rename(tmp_file, new_file)
vsi_stat(new_file)
vsi_unlink(new_file)
```

vsi_rmdir *Delete a directory*

Description

vsi_rmdir() deletes a directory object from the file system. On some systems the directory must be empty before it can be deleted. With recursive = TRUE, deletes a directory object and its content from the file system. This function goes through the GDAL VSIFileHandler virtualization and may work on unusual filesystems such as in memory. It is a wrapper for VSIRmdir() and VSIRmdirRecursive() in the GDAL Common Portability Library.

Usage

```
vsi_rmdir(path, recursive = FALSE)
```

Arguments

path	Character string. The path to the directory to be deleted.
recursive	Logical scalar. TRUE to delete the directory and its content. Defaults to FALSE.

Value

0 on success or -1 on an error.

Note

/vsis3/ has an efficient implementation for deleting recursively. Starting with GDAL 3.4, /vsigs/ has an efficient implementation for deleting recursively, provided that OAuth2 authentication is used.

See Also

deleteDataset(), vsi_mkdir(), vsi_read_dir(), vsi_unlink()

Examples

```
new_dir <- file.path(tempdir(), "newdir")
vsi_mkdir(new_dir)
vsi_rmdir(new_dir)
```

`vsi_set_path_option` *Set a path specific option for a given path prefix*

Description

`vsi_set_path_option()` sets a path specific option for a given path prefix. Such an option is typically, but not limited to, setting credentials for a virtual file system. Wrapper for `VSISetPathSpecificOption()` in the GDAL Common Portability Library. Requires GDAL >= 3.6.

Usage

```
vsi_set_path_option(path_prefix, key, value)
```

Arguments

`path_prefix`	Character string. A path prefix of a virtual file system handler. Typically of the form `/vsiXXX/bucket`.
`key`	Character string. Option key.
`value`	Character string. Option value. Passing `value = ""` (empty string) will unset a value previously set by `vsi_set_path_option()`.

Details

Options may also be set with `set_config_option()`, but `vsi_set_path_option()` allows specifying them with a granularity at the level of a file path. This makes it easier if using the same virtual file system but with different credentials (e.g., different credentials for buckets "/vsis3/foo" and "/vsis3/bar"). This is supported for the following virtual file systems: /vsis3/, /vsigs/, /vsiaz/, /vsioss/, /vsiwebhdfs, /vsiswift.

Value

No return value, called for side effect.

Note

Setting options for a path starting with /vsiXXX/ will also apply for /vsiXXX_streaming/ requests.

No particular care is taken to store options in RAM in a secure way. So they might accidentally hit persistent storage if swapping occurs, or someone with access to the memory allocated by the process may be able to read them.

See Also

`set_config_option()`, `vsi_clear_path_options()`

vsi_stat *Get filesystem object info*

Description

These functions work on GDAL virtual file systems such as in-memory (/vsimem/), URLs (/vsicurl/), cloud storage services (e.g., /vsis3/, /vsigs/, /vsiaz/, etc.), compressed archives (e.g., /vsizip, /vsitar/, /vsi7z/, /vsigzip/, etc.), and others including "standard" file systems. See https://gdal.org/en/stable/user/virtual_file_systems.html.

Usage

```
vsi_stat(filename, info = "exists")

vsi_stat_exists(filenames)

vsi_stat_type(filenames)

vsi_stat_size(filenames)
```

Arguments

filename	Character string. The path of the filesystem object to be queried.
info	Character string. The type of information to fetch, one of "exists" (the default), "type" or "size".
filenames	Character vector of filesystem objects to query.

Details

vsi_stat() fetches status information about a single filesystem object (file, directory, etc). It is a wrapper for VSIStatExL() in the GDAL Common Portability Library. Analog of the POSIX stat() function.

vsi_stat_exists(), vsi_stat_type() and vsi_stat_size() are specializations operating on a vector of potentially multiple file system object names, returning, respectfully, a logical vector, a character vector, and a numeric vector carrying the bit64::integer64 class attribute.

Value

If info = "exists", vsi_stat() returns logical TRUE if the file system object exists, otherwise FALSE. If info = "type", returns a character string with one of "file" (regular file), "dir" (directory), "symlink" (symbolic link), or empty string (""). If info = "size", returns the file size in bytes (as bit64::integer64 type), or -1 if an error occurs. vsi_stat_exists() returns a logical vector. vsi_stat_type() returns a character vector. vsi_stat_size() returns a numeric vector carrying the bit64::integer64 class attribute.

Note

For portability, `vsi_stat()` supports a subset of `stat()`-type information for filesystem objects. This function is primarily intended for use with GDAL virtual file systems (e.g., URLs, cloud storage systems, ZIP/GZip/7z/RAR archives, in-memory files), but can also be used on "standard" file systems (e.g., in the / hierarchy on Unix-like systems or in C:, D:, etc. drives on Windows).

See Also

GDAL Virtual File Systems:
https://gdal.org/en/stable/user/virtual_file_systems.html

Examples

```
data_dir <- system.file("extdata", package="gdalraster")
vsi_stat(data_dir)
vsi_stat(data_dir, "type")
# stat() on a directory doesn't return the sum of the file sizes in it,
# but rather how much space is used by the directory entry
vsi_stat(data_dir, "size")

elev_file <- file.path(data_dir, "storml_elev.tif")
vsi_stat(elev_file)
vsi_stat(elev_file, "type")
vsi_stat(elev_file, "size")

nonexistent <- file.path(data_dir, "nonexistent.tif")
vsi_stat(nonexistent)
vsi_stat(nonexistent, "type")
vsi_stat(nonexistent, "size")

fs_objects <- c(data_dir, elev_file, nonexistent)
vsi_stat_exists(fs_objects)
vsi_stat_type(fs_objects)
vsi_stat_size(fs_objects)

# /vsicurl/ file system handler
base_url <- "https://raw.githubusercontent.com/usdaforestservice/"
f <- "gdalraster/main/sample-data/landsat_c2ard_sr_mt_hood_jul2022_utm.tif"
url_file <- paste0("/vsicurl/", base_url, f)

# try to be CRAN-compliant for the example:
set_config_option("GDAL_HTTP_CONNECTTIMEOUT", "10")
set_config_option("GDAL_HTTP_TIMEOUT", "10")

vsi_stat(url_file)
vsi_stat(url_file, "type")
vsi_stat(url_file, "size")
```

vsi_supports_rnd_write *Return whether the filesystem supports random write*

Description

`vsi_supports_rnd_write()` returns whether the filesystem supports random write. Wrapper for `VSISupportsRandomWrite()` in the GDAL API.

Usage

```
vsi_supports_rnd_write(filename, allow_local_tmpfile)
```

Arguments

`filename`	Character string. The path of the filesystem object to be tested.
`allow_local_tmpfile`	Logical scalar. `TRUE` if the filesystem is allowed to use a local temporary file before uploading to the target location.

Value

Logical scalar. `TRUE` if random write is supported.

Note

The location GDAL uses for temporary files can be forced via the `CPL_TMPDIR` configuration option.

See Also

`vsi_supports_seq_write()`

Examples

```
# Requires GDAL >= 3.6
if (gdal_version_num() >= gdal_compute_version(3, 6, 0))
  vsi_supports_rnd_write("/vsimem/test-mem-file.gpkg", TRUE)
```

vsi_supports_seq_write *Return whether the filesystem supports sequential write*

Description

`vsi_supports_seq_write()` returns whether the filesystem supports sequential write. Wrapper for `VSISupportsSequentialWrite()` in the GDAL API.

Usage

```
vsi_supports_seq_write(filename, allow_local_tmpfile)
```

Arguments

`filename`	Character string. The path of the filesystem object to be tested.
`allow_local_tmpfile`	Logical scalar. `TRUE` if the filesystem is allowed to use a local temporary file before uploading to the target location.

Value

Logical scalar. `TRUE` if sequential write is supported.

Note

The location GDAL uses for temporary files can be forced via the `CPL_TMPDIR` configuration option.

See Also

`vsi_supports_rnd_write()`

Examples

```
# Requires GDAL >= 3.6
if (gdal_version_num() >= gdal_compute_version(3, 6, 0))
  vsi_supports_seq_write("/vsimem/test-mem-file.gpkg", TRUE)
```

vsi_sync *Synchronize a source file/directory with a target file/directory*

Description

vsi_sync() is a wrapper for VSISync() in the GDAL Common Portability Library. The GDAL documentation is given in Details.

Usage

```
vsi_sync(src, target, show_progress = FALSE, options = NULL)
```

Arguments

src	Character string. Source file or directory.
target	Character string. Target file or directory.
show_progress	Logical scalar. If TRUE, a progress bar will be displayed. Defaults to FALSE.
options	Character vector of NAME=VALUE pairs (see Details).

Details

VSISync() is an analog of the Linux rsync utility. In the current implementation, rsync would be more efficient for local file copying, but VSISync() main interest is when the source or target is a remote file system like /vsis3/ or /vsigs/, in which case it can take into account the timestamps of the files (or optionally the ETag/MD5Sum) to avoid unneeded copy operations. This is only implemented efficiently for:

- local filesystem <–> remote filesystem
- remote filesystem <–> remote filesystem (starting with GDAL 3.1)
  Where the source and target remote filesystems are the same and one of /vsis3/, /vsigs/ or /vsiaz/. Or when the target is /vsiaz/ and the source is /vsis3/, /vsigs/, /vsiadls/ or /vsicurl/ (starting with GDAL 3.8)

Similarly to rsync behavior, if the source filename ends with a slash, it means that the content of the directory must be copied, but not the directory name. For example, assuming "/home/even/foo" contains a file "bar", VSISync("/home/even/foo/", "/mnt/media", ...) will create a "/mnt/media/bar" file. Whereas VSISync("/home/even/foo", "/mnt/media", ...) will create a "/mnt/media/foo" directory which contains a bar file.

The options argument accepts a character vector of name=value pairs. Currently accepted options are:

- RECURSIVE=NO (the default is YES)

- SYNC_STRATEGY=TIMESTAMP/ETAG/OVERWRITE. Determines which criterion is used to determine if a target file must be replaced when it already exists and has the same file size as the source. Only applies for a source or target being a network filesystem. The default is TIMESTAMP (similarly to how 'aws s3 sync' works), that is to say that for an upload operation, a remote file is replaced if it has a different size or if it is older than the source. For a download operation, a local file is replaced if it has a different size or if it is newer than the remote file. The ETAG strategy assumes that the ETag metadata of the remote file is the MD5Sum of the file content, which is only true in the case of /vsis3/ for files not using KMS server side encryption and uploaded in a single PUT operation (so smaller than 50 MB given the default used by GDAL). Only to be used for /vsis3/, /vsigs/ or other filesystems using a MD5Sum as ETAG. The OVERWRITE strategy (GDAL >= 3.2) will always overwrite the target file with the source one.

- NUM_THREADS=integer. Number of threads to use for parallel file copying. Only use for when /vsis3/, /vsigs/, /vsiaz/ or /vsiadls/ is in source or target. The default is 10 since GDAL 3.3.

- CHUNK_SIZE=integer. Maximum size of chunk (in bytes) to use to split large objects when downloading them from /vsis3/, /vsigs/, /vsiaz/ or /vsiadls/ to local file system, or for upload to /vsis3/, /vsiaz/ or /vsiadls/ from local file system. Only used if NUM_THREADS > 1. For upload to /vsis3/, this chunk size must be set at least to 5 MB. The default is 8 MB since GDAL 3.3.

- x-amz-KEY=value. (GDAL >= 3.5) MIME header to pass during creation of a /vsis3/ object.

- x-goog-KEY=value. (GDAL >= 3.5) MIME header to pass during creation of a /vsigs/ object.

- x-ms-KEY=value. (GDAL >= 3.5) MIME header to pass during creation of a /vsiaz/ or /vsiadls/ object.

Value

Logical scalar, TRUE on success or FALSE on an error.

See Also

copyDatasetFiles(), vsi_copy_file()

Examples

```
## Not run:
# sample-data is a directory in the git repository for gdalraster that is
# not included in the R package:
# https://github.com/USDAForestService/gdalraster/tree/main/sample-data
# A copy of sample-data in an AWS S3 bucket, and a partial copy in an
# Azure Blob container, were used to generate the example below.

src <- "/vsis3/gdalraster-sample-data/"
# s3://gdalraster-sample-data is not public, set credentials
set_config_option("AWS_ACCESS_KEY_ID", "xxxxxxxxxxxxx")
set_config_option("AWS_SECRET_ACCESS_KEY", "xxxxxxxxxxxxxxxxxxxxxxxxxxxxx")
```

```
vsi_read_dir(src)
#> [1] "README.md"
#> [2] "bl_mrbl_ng_jul2004_rgb_720x360.tif"
#> [3] "blue_marble_ng_neo_metadata.xml"
#> [4] "landsat_c2ard_sr_mt_hood_jul2022_utm.json"
#> [5] "landsat_c2ard_sr_mt_hood_jul2022_utm.tif"
#> [6] "lf_elev_220_metadata.html"
#> [7] "lf_elev_220_mt_hood_utm.tif"
#> [8] "lf_fbfm40_220_metadata.html"
#> [9] "lf_fbfm40_220_mt_hood_utm.tif"

dst <- "/vsiaz/sampledata"
set_config_option("AZURE_STORAGE_CONNECTION_STRING",
                  "<connection_string_for_gdalraster_account>")
vsi_read_dir(dst)
#> [1] "lf_elev_220_metadata.html"   "lf_elev_220_mt_hood_utm.tif"

# GDAL VSISync() supports direct copy for /vsis3/ -> /vsiaz/ (GDAL >= 3.8)
result <- vsi_sync(src, dst, show_progress = TRUE)
#> 0...10...20...30...40...50...60...70...80...90...100 - done.
print(result)
#> [1] TRUE
vsi_read_dir(dst)
#> [1] "README.md"
#> [2] "bl_mrbl_ng_jul2004_rgb_720x360.tif"
#> [3] "blue_marble_ng_neo_metadata.xml"
#> [4] "landsat_c2ard_sr_mt_hood_jul2022_utm.json"
#> [5] "landsat_c2ard_sr_mt_hood_jul2022_utm.tif"
#> [6] "lf_elev_220_metadata.html"
#> [7] "lf_elev_220_mt_hood_utm.tif"
#> [8] "lf_fbfm40_220_metadata.html"
#> [9] "lf_fbfm40_220_mt_hood_utm.tif"

## End(Not run)
```

`vsi_unlink` *Delete a file*

Description

`vsi_unlink()` deletes a file object from the file system. This function goes through the GDAL `VSIFileHandler` virtualization and may work on unusual filesystems such as in memory. It is a wrapper for `VSIUnlink()` in the GDAL Common Portability Library. Analog of the POSIX `unlink()` function.

Usage

```
vsi_unlink(filename)
```

Arguments

filename	Character string. The path of the file to be deleted.

Value

0 on success or -1 on an error.

See Also

deleteDataset(), vsi_rmdir(), vsi_unlink_batch()

Examples

```
elev_file <- system.file("extdata/storml_elev.tif", package="gdalraster")
mem_file <- file.path("/vsimem", "tmp.tif")
copyDatasetFiles(mem_file, elev_file)
vsi_read_dir("/vsimem")
vsi_unlink(mem_file)
vsi_read_dir("/vsimem")
```

---

vsi_unlink_batch *Delete several files in a batch*

Description

vsi_unlink_batch() deletes a list of files passed in a character vector. All files should belong to the same file system handler. This is implemented efficiently for /vsis3/ and /vsigs/ (provided for /vsigs/ that OAuth2 authentication is used). This function is a wrapper for VSIUnlinkBatch() in the GDAL Common Portability Library.

Usage

```
vsi_unlink_batch(filenames)
```

Arguments

filenames	Character vector. The list of files to delete.

Value

Logical vector of length(filenames) with values depending on the success of deletion of the corresponding file. NULL might be returned in case of a more general error (for example, files belonging to different file system handlers).

See Also

deleteDataset(), vsi_rmdir(), vsi_unlink()

Examples

```
# regular file system for illustration
elev_file <- system.file("extdata/storml_elev.tif", package="gdalraster")
tcc_file <- system.file("extdata/storml_tcc.tif", package="gdalraster")

tmp_elev <- file.path(tempdir(), "tmp_elev.tif")
file.copy(elev_file, tmp_elev)
tmp_tcc <- file.path(tempdir(), "tmp_tcc.tif")
file.copy(tcc_file, tmp_tcc)
vsi_unlink_batch(c(tmp_elev, tmp_tcc))
```

---

warp *Raster reprojection and mosaicing*

Description

`warp()` is a wrapper of the `gdalwarp` command-line utility for raster reprojection and warping (see https://gdal.org/en/stable/programs/gdalwarp.html). The function can reproject to any supported spatial reference system (SRS). It can also be used to crop, mosaic, resample, and optionally write output to a different raster format. See Details for a list of commonly used processing options that can be passed as arguments to `warp()`.

Usage

```
warp(src_files, dst_filename, t_srs, cl_arg = NULL, quiet = FALSE)
```

Arguments

`src_files`	Either a character vector of source filenames(s) to be reprojected, or a `GDALRaster` object or list of `GDALRaster` objects for the source data.
`dst_filename`	Either a character string giving the filename of the output dataset, or an object of class `GDALRaster` for the output.
`t_srs`	Character string. Target spatial reference system. Usually an EPSG code ("EPSG:#####") or a well known text (WKT) SRS definition. Can be set to empty string `""` and the spatial reference of `src_files[1]` will be used unless the destination raster already exists (see Note).
`cl_arg`	Optional character vector of command-line arguments to `gdalwarp` in addition to `-t_srs` (see Details).
`quiet`	Logical scalar. If `TRUE`, a progress bar will not be displayed. Defaults to `FALSE`.

Details

Several processing options can be performed in one call to `warp()` by passing the necessary command-line arguments. The following list describes several commonly used arguments. Note that `gdalwarp` supports a large number of arguments that enable a variety of different processing options. Users are encouraged to review the original source documentation provided by the GDAL project at the URL above for the full list.

- -te <xmin> <ymin> <xmax> <ymax>
  Georeferenced extents of output file to be created (in target SRS by default).
- -te_srs <srs_def>
  SRS in which to interpret the coordinates given with -te (if different than t_srs).
- -tr <xres> <yres>
  Output pixel resolution (in target georeferenced units).
- -tap
  (target aligned pixels) align the coordinates of the extent of the output file to the values of the -tr, such that the aligned extent includes the minimum extent. Alignment means that xmin / resx, ymin / resy, xmax / resx and ymax / resy are integer values.
- -ovr <level>|AUTO|AUTO-<n>|NONE
  Specify which overview level of source files must be used. The default choice, AUTO, will select the overview level whose resolution is the closest to the target resolution. Specify an integer value (0-based, i.e., 0=1st overview level) to select a particular level. Specify AUTO-n where n is an integer greater or equal to 1, to select an overview level below the AUTO one. Or specify NONE to force the base resolution to be used (can be useful if overviews have been generated with a low quality resampling method, and the warping is done using a higher quality resampling method).
- -wo <NAME>=<VALUE>
  Set a warp option as described in the GDAL documentation for GDALWarpOptions Multiple -wo may be given. See also -multi below.
- -ot <type>
  Force the output raster bands to have a specific data type supported by the format, which may be one of the following: Byte, Int8, UInt16, Int16, UInt32, Int32, UInt64, Int64, Float32, Float64, CInt16, CInt32, CFloat32 or CFloat64.
- -r <resampling_method>
  Resampling method to use. Available methods are: near (nearest neighbour, the default), bilinear, cubic, cubicspline, lanczos, average, rms (root mean square, GDAL >= 3.3), mode, max, min, med, q1 (first quartile), q3 (third quartile), sum (GDAL >= 3.1).
- -srcnodata "<value>[<value>]..."
  Set nodata masking values for input bands (different values can be supplied for each band). If more than one value is supplied all values should be quoted to keep them together as a single operating system argument. Masked values will not be used in interpolation. Use a value of None to ignore intrinsic nodata settings on the source dataset. If -srcnodata is not explicitly set, but the source dataset has nodata values, they will be taken into account by default.
- -dstnodata "<value>[<value>]..."
  Set nodata values for output bands (different values can be supplied for each band). If more than one value is supplied all values should be quoted to keep them together as a single operating system argument. New files will be initialized to this value and if possible the nodata value will be recorded in the output file. Use a value of "None" to ensure that nodata is not defined. If this argument is not used then nodata values will be copied from the source dataset.
- -srcband <n>
  (GDAL >= 3.7) Specify an input band number to warp (between 1 and the number of bands of the source dataset). This option is used to warp a subset of the input bands. All input bands are used when it is not specified. This option may be repeated multiple times to select several input bands. The order in which bands are specified will be the order in which they appear in

the output dataset (unless -dstband is specified). The alpha band should not be specified in the list, as it will be automatically retrieved (unless -nosrcalpha is specified).

- -dstband <n>
(GDAL >= 3.7) Specify the output band number in which to warp. In practice, this option is only useful when updating an existing dataset, e.g to warp one band at at time. If -srcband is specified, there must be as many occurrences of -dstband as there are of -srcband.
If -dstband is not specified, then:
c("-dstband", "1", "-dstband", "2", ... "-dstband", "N")
is assumed where N is the number of input bands (implicitly, or specified explicitly with -srcband). The alpha band should not be specified in the list, as it will be automatically retrieved (unless -nosrcalpha is specified).
- -wm <memory_in_mb>
Set the amount of memory that the warp API is allowed to use for caching. The value is interpreted as being in megabytes if the value is <10000. For values >=10000, this is interpreted as bytes. The warper will total up the memory required to hold the input and output image arrays and any auxiliary masking arrays and if they are larger than the "warp memory" allowed it will subdivide the chunk into smaller chunks and try again. If the -wm value is very small there is some extra overhead in doing many small chunks so setting it larger is better but it is a matter of diminishing returns.
- -multi
Use multithreaded warping implementation. Two threads will be used to process chunks of image and perform input/output operation simultaneously. Note that computation is not multithreaded itself. To do that, you can use the -wo NUM_THREADS=val/ALL_CPUS option, which can be combined with -multi.
- -of <format> Set the output raster format. Will be guessed from the extension if not specified. Use the short format name (e.g., "GTiff").
- -co <NAME>=<VALUE>
Set one or more format specific creation options for the output dataset. For example, the GeoTIFF driver supports creation options to control compression, and whether the file should be tiled. getCreationOptions() can be used to look up available creation options, but the GDAL Raster drivers documentation is the definitive reference for format specific options. Multiple -co may be given, e.g.,

```
c("-co", "COMPRESS=LZW", "-co", "BIGTIFF=YES")
```

- -overwrite
Overwrite the target dataset if it already exists. Overwriting means deleting and recreating the file from scratch. Note that if this option is not specified and the output file already exists, it will be updated in place.

The documentation for gdalwarp describes additional command-line options related to spatial reference systems, alpha bands, masking with polygon cutlines including blending, and more.

Mosaicing into an existing output file is supported if the output file already exists. The spatial extent of the existing file will not be modified to accommodate new data, so you may have to remove it in that case, or use the -overwrite option.

Command-line options are passed to warp() as a character vector. The elements of the vector are the individual options followed by their individual values, e.g.,

```
cl_arg = c("-tr", "30", "30", "-r", "bilinear"))
```

to set the target pixel resolution to 30 x 30 in target georeferenced units and use bilinear resampling.

Value

Logical indicating success (invisible TRUE). An error is raised if the operation fails.

Note

warp() can be used to reproject and also perform other processing such as crop, resample, and mosaic. This processing is generally done with a single function call by passing arguments for the output ("target") pixel resolution, extent, resampling method, nodata value, format, and so forth.

If warp() is called with t_srs = "" and the output raster does not already exist, the target spatial reference will be set to that of src_files[1]. In that case, the processing options given in cl_arg will be performed without reprojecting (if there is one source raster or multiple sources that all use the same spatial reference system, otherwise would reproject inputs to the SRS of src_files[1] where they are different). If t_srs = "" and the destination raster already exists, the output SRS will be the projection of the destination dataset.

See Also

GDALRaster-class, srs_to_wkt(), translate()

Examples

```
# reproject the elevation raster to NAD83 / CONUS Albers (EPSG:5070)
elev_file <- system.file("extdata/storml_elev.tif", package="gdalraster")

# command-line arguments for gdalwarp
# resample to 90-m resolution and keep pixels aligned:
args <- c("-tr", "90", "90", "-r", "cubic", "-tap")
# write to Erdas Imagine format (HFA) with compression:
args <- c(args, "-of", "HFA", "-co", "COMPRESSED=YES")

alb83_file <- file.path(tempdir(), "storml_elev_alb83.img")
warp(elev_file, alb83_file, t_srs = "EPSG:5070", cl_arg = args)

ds <- new(GDALRaster, alb83_file)
ds$getDriverLongName()
ds$getProjectionRef()
ds$res()
ds$getStatistics(band=1, approx_ok=FALSE, force=TRUE)
ds$close()
```

Index