Package ‘sf’

May 24, 2023

Version 1.0-13

Title Simple Features for R

Description Support for simple features, a standardized way to encode spatial vector data. Binds to 'GDAL' for reading and writing data, to 'GEOS' for geometrical operations, and to 'PROJ' for projection conversions and datum transformations. Uses by default the 's2' package for spherical geometry operations on ellipsoidal (long/lat) coordinates.

License GPL-2 | MIT + file LICENSE


BugReports https://github.com/r-spatial/sf/issues/

Depends methods, R (>= 3.3.0)

Imports classInt (>= 0.4-1), DBI (>= 0.8), graphics, grDevices, grid, magrittr, Rcpp (>= 0.12.18), s2 (>= 1.1.0), stats, tools, units (>= 0.7-0), utils

Suggests blob, covr, dplyr (>= 0.8-3), ggplot2, knitr, lwgeom (>= 0.2-1), maps, mapview, Matrix, microbenchmark, odbc, pbapply, pillar, pool, raster, rlang, rmarkdown, RPostgres (>= 1.1.0), RPostgreSQL, RSQLite, RSQLite, spatstat (>= 2.0-1), spatstat.geom, spatstat.random, spatstat.linnet, spatstat.utils, stars (>= 0.2-0), terra, testthat (>= 3.0.0), tibble (>= 1.4.1), tidyrr (>= 1.2.0), tidyselect (>= 1.0.0), tmap (>= 2.0), vctrs, wk

LinkingTo Rcpp

VignetteBuilder knitr

Encoding UTF-8

RoxygenNote 7.2.3

Config/testthat/edition 3

SystemRequirements GDAL (>= 2.0.1), GEOS (>= 3.4.0), PROJ (>= 4.8.0), sqlite3
# R topics documented:

Collate 'RcppExports.R' 'init.R' 'crs.R' 'bbox.R' 'read.R' 'db.R'
'sfc.R' 'sf.R' 'bbox.R' 'wkb.R' 'wkt.R' 'plot.R'
'geom-measures.R' 'geom-predicates.R' 'geom-transformers.R'
'transform.R' 'proj.R' 'sp.R' 'grid.R' 'arith.R' 'tidyverse.R'
'tidyverse-vctrs.R' 'cast_sfg.R' 'cast_sfc.R' 'graticule.R'
'datasets.R' 'aggregate.R' 'agr.R' 'maps.R' 'join.R' 'sample.R'
'valid.R' 'collection_extract.R' 'jitter.R' 'gbq.R'
'spatstat.R' 'stars.R' 'crop.R' 'gdal_utils.R' 'nearest.R'
'normalize.R' 'defunct.R' 'x_range.R' 'm_range.R'
'shift_longitude.R' 'make_grid.R' 's2.R' 'terra.R'
'geos-overlating.R' 'break_antimeridian.R'

## NeedsCompilation
yes

**Author**  Edzer Pebesma [aut, cre] (<https://orcid.org/0000-0001-8049-7069>),
Roger Bivand [ctb] (<https://orcid.org/0000-0003-2392-6140>),
Etienne Racine [ctb],
Michael Sumner [ctb],
Ian Cook [ctb],
Tim Keitt [ctb],
Robin Lovelace [ctb],
Hadley Wickham [ctb],
Jeroen Ooms [ctb] (<https://orcid.org/0000-0002-4035-0289>),
Kirill Müller [ctb],
Thomas Lin Pedersen [ctb],
Dan Baston [ctb],
Dewey Dunnington [ctb] (<https://orcid.org/0000-0002-9415-4582>)

**Maintainer**  Edzer Pebesma <edzer.pebesma@uni-muenster.de>

**Repository**  CRAN

**Date/Publication**  2023-05-24 08:10:02 UTC

## R topics documented:

- aggregate.sf ................................................. 4
- as .......................................................... 6
- bind ........................................................ 7
- dbDataType,PostgreSQLConnection,sf-method .................. 8
- dbWriteTable,PostgreSQLConnection,character,sf-method .... 8
- db_drivers ................................................ 10
- extension_map ............................................. 10
- gdal ....................................................... 10
- gdal_addo ............................................... 14
- gdal_utils ............................................... 15
- geos_binary_ops ......................................... 17
- geos_binary_pred ......................................... 19
- geos_combine ............................................ 22
- geos_measures ........................................... 23
- geos_query ............................................... 25
aggregate an sf object

Description
aggregate an sf object, possibly union-ing geometries

Usage
## S3 method for class 'sf'
aggregate(
  x,
  by,
  FUN,
  ..., 
  do_union = TRUE,
  simplify = TRUE,
  join = st_intersects
)

Arguments

- x: object of class sf
- by: either a list of grouping vectors with length equal to nrow(x) (see aggregate), or an object of class sf or sfc with geometries that are used to generate groupings, using the binary predicate specified by the argument join
aggregate.sf

FUN  function passed on to aggregate, in case ids was specified and attributes need to be grouped
... arguments passed on to FUN
do_union  logical; should grouped geometries be unioned using st_union? See details.
simplify  logical; see aggregate
join  logical spatial predicate function to use if by is a simple features object or geometry; see st_join

Details

In case do_union is FALSE, aggregate will simply combine geometries using c.sfg. When polygons sharing a boundary are combined, this leads to geometries that are invalid; see https://github.com/r-spatial/sf/issues/681.

Value

an sf object with aggregated attributes and geometries; additional grouping variables having the names of names(ids) or are named Group.i for ids[[i]]; see aggregate.

Note

Does not work using the formula notation involving ~ defined in aggregate.

Examples

m1 = cbind(c(0, 0, 1, 0), c(0, 1, 1, 0))
m2 = cbind(c(0, 1, 1, 0), c(0, 0, 1, 0))
pol = st_sfc(st_polygon(list(m1)), st_polygon(list(m2)))
set.seed(1985)
d = data.frame(matrix(runif(15), ncol = 3))
p = st_as_sf(x = d, coords = 1:2)
plot(pol)
plot(p, add = TRUE)
(p_ag1 = aggregate(p, pol, mean))
plot(p_ag1)  # geometry same as pol
# works when x overlaps multiple objects in 'by':
p_buff = st_buffer(p, 0.2)
plot(p_buff, add = TRUE)
(p_ag2 = aggregate(p_buff, pol, mean))  # increased mean of second
# with non-matching features
m3 = cbind(c(0, 0, -0.1, 0), c(0, 0.1, 0.1, 0))
pol = st_sfc(st_polygon(list(m3)), st_polygon(list(m1)), st_polygon(list(m2)))
(p_ag3 = aggregate(p, pol, mean))
plot(p_ag3)
# In case we need to pass an argument to the join function:
(p_ag4 = aggregate(p, pol, mean,
    join = function(x, y) st_is_within_distance(x, y, dist = 0.3)))
Methods to coerce simple features to Spatial* and Spatial*DataFrame objects

Description

`as_Spatial()` allows to convert sf and sfc to Spatial*DataFrame and Spatial* for sp compatibility. You can also use `as(x, "Spatial")` to transform sp objects to sf and sfc with `as(x, "sf")`.

Usage

```r
as_Spatial(from, cast = TRUE, IDs = paste0("ID", seq_along(from)))
```

Arguments

- `from`: object of class sf, sfc_POINT, sfc_MULTIPOINT, sfc_LINESTRING, sfc_MULTILINESTRING, sfc_POLYGON, or sfc_MULTIPOLYGON.
- `cast`: logical; if TRUE, `st_cast()` from before converting, so that e.g. GEOMETRY objects with a mix of POLYGON and MULTIPOLYGON are cast to MULTIPOLYGON.
- `IDs`: character vector with IDs for the Spatial* geometries

Details

Package sp supports three dimensions for POINT and MULTIPoint (SpatialPoint*). Other geometries must be two-dimensional (XY). Dimensions can be dropped using `st_zm()` with what = "M" or what = "ZM".

For converting simple features (i.e., sf objects) to their Spatial counterpart, use `as(obj, "Spatial")`

Value

geometry-only object deriving from Spatial, of the appropriate class

Examples

```r
nc <- st_read(system.file("shape/nc.shp", package="sf"))
if (require(sp, quietly = TRUE)) {
  # convert to SpatialPolygonsDataFrame
  spdf <- as_Spatial(nc)
  # identical to
  spdf <- as(nc, "Spatial")
  # convert to SpatialPolygons
  as(st_geometry(nc), "Spatial")
  # back to sf
  as(spdf, "sf")
}
```
bind

**Bind rows (features) of sf objects**

**Description**

Bind rows (features) of sf objects

Bind columns (variables) of sf objects

**Usage**

```r
## S3 method for class 'sf'
rbind(...., deparse.level = 1)

## S3 method for class 'sf'
cbind(...., deparse.level = 1, sf_column_name = NULL)

st_bind_cols(...)
```

**Arguments**

... objects to bind; note that for the rbind and cbind methods, all objects have to be of class sf; see dotsMethods
deparse.level integer; see rbind
sf_column_name character; specifies active geometry; passed on to st_sf

**Details**

both rbind and cbind have non-standard method dispatch (see cbind): the rbind or cbind method for sf objects is only called when all arguments to be binded are of class sf.
If you need to cbind e.g. a data.frame to an sf, use data.frame directly and use st_sf on its result, or use bind_cols; see examples.
st_bind_cols is deprecated; use cbind instead.

**Value**

cbind called with multiple sf objects warns about multiple geometry columns present when the geometry column to use is not specified by using argument sf_column_name; see also st_sf.

**Examples**

```r
crs = st_crs(3857)
a = st_sf(a=1, geom = st_sfc(st_point(0:1)), crs = crs)
b = st_sf(a=1, geom = st_sfc(st_linestring(matrix(1:4,2))), crs = crs)
c = st_sf(a=4, geom = st_sfc(st_multilinestring(list(matrix(1:4,2)))), crs = crs)
rbind(a,b,c)
rbind(a,b)
```
rbind(a, b)
rbind(b, c)
cbind(a, b, c) # warns
if (require(dplyr, quietly = TRUE))
  dplyr::bind_cols(a, b)
c = st_sf(a=4, geomc = st_sfc(st_multilinestring(list(matrix(1:4,2)))), crs = crs)
cbind(a, b, c, sf_column_name = "geomc")
df = data.frame(x=3)
st_sf(data.frame(c, df))
if (require(dplyr, quietly = TRUE))
  dplyr::bind_cols(c, df)

### S4 method for signature 'PostgreSQLConnection,sf'
dbDataType(dbObj, obj)

### S4 method for signature 'DBIOobject,sf'
dbDataType(dbObj, obj)

Arguments

- **dbObj** (DBIOobject driver or connection)
- **obj** (Object to convert)

---

Write sf object to Database

- **Description**
  Write sf object to Database
  Write sf object to Database
Usage

```r
## S4 method for signature 'PostgreSQLConnection,character,sf'
dbWriteTable(
  conn,
  name,
  value,
  ...
)

## S4 method for signature 'DBIOObject,character,sf'
dbWriteTable(
  conn,
  name,
  value,
  ...
)
```

Arguments

- `conn` DBIOObject
- `name` character vector of names (table names, fields, keywords).
- `value` a data.frame.
- `row.names` Add a row.name column, or a vector of length nrow(obj) containing row.names; default FALSE.
- `overwrite` Will try to drop table before writing; default FALSE.
- `append` Append rows to existing table; default FALSE.
- `field.types` default NULL. Allows to override type conversion from R to PostgreSQL. See dbDataType() for details.
- `binary` Send geometries serialized as Well-Known Binary (WKB); if FALSE, uses Well-Known Text (WKT). Defaults to TRUE (WKB).
### db_drivers

*Drivers for which update should be TRUE by default*

**Description**

Drivers for which update should be TRUE by default

**Usage**

```r
db_drivers
```

**Format**

An object of class character of length 12.

---

### extension_map

*Map extension to driver*

**Description**

Map extension to driver

**Usage**

```r
extension_map
```

**Format**

An object of class list of length 26.

---

### gdal

*functions to interact with gdal not meant to be called directly by users (but e.g. by stars::read_stars)*

**Description**

functions to interact with gdal not meant to be called directly by users (but e.g. by stars::read_stars)
Usage

gdal_read(
  x,
  ..., 
  options = character(0),
  driver = character(0),
  read_data = TRUE,
  NA_value = NA_real_,
  RasterIO_parameters = list()
)

gdal_write(
  x,
  ..., 
  file,
  driver = "GTiff",
  options = character(0),
  type = "Float32",
  NA_value = NA_real_,
  geotransform,
  update = FALSE,
  scale_offset = c(1, 0)
)

gdal_inv_geotransform(gt)

gdal_crs(file, options = character(0))

gdal_metadata(
  file,
  domain_item = character(0),
  options = character(0),
  parse = TRUE
)

gdal_subdatasets(file, options = character(0), name = TRUE)

gdal_polygonize(
  x,
  mask = NULL,
  file = tempfile(),
  driver = "GTiff",
  use_integer = TRUE,
  geotransform,
  breaks = classInt::classIntervals(na.omit(as.vector(x[[1]])))$brks,
  use_contours = FALSE,
  contour_lines = FALSE,
  connect8 = FALSE,
gdal_rasterize(sf, x, gt, file, driver = "GTiff", options = character())

gdal_extract(f, pts, bilinear = FALSE)

gdal_read_mdim(
  file,
  array_name = character(0),
  options = character(0),
  offset = integer(0),
  count = integer(0),
  step = integer(0),
  proxy = FALSE,
  debug = FALSE
)

gdal_write_mdim(
  file,
  driver,
  dimx,
  cdl,
  wkt,
  xy,
  ...,
  root_group_options = character(0),
  options = character(0),
  as_float = TRUE
)

gdal_create(f, nxy, values, crs, xlim, ylim)

Arguments

x character vector, possibly of length larger than 1 when more than one raster is read
...
options character; driver specific options regarding reading or creating the dataset
driver character; driver short name; when empty vector, driver is auto-detected.
read_data logical; if FALSE, only the imagery metadata is returned
NA_value (double) non-NA value to use for missing values; if NA, when writing missing values are not specially flagged in output dataset, when reading the default (dataset) missing values are used (if present / set).

RasterIO_parameters
list with named parameters to GDAL’s RasterIO; see the stars::read_stars documentation.
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>file name</td>
</tr>
<tr>
<td>type</td>
<td>gdal write type</td>
</tr>
<tr>
<td>geotransform</td>
<td>length 6 numeric vector with GDAL geotransform parameters.</td>
</tr>
<tr>
<td>update</td>
<td>logical; TRUE if in an existing raster file pixel values shall be updated.</td>
</tr>
<tr>
<td>scale_offset</td>
<td>length 2 numeric; contains scale and offset values</td>
</tr>
<tr>
<td>gt</td>
<td>double vector of length 6</td>
</tr>
<tr>
<td>domain_item</td>
<td>character vector of length 0, 1 (with domain), or 2 (with domain and item); use &quot;&quot; for the default domain, use NA_character_ to query the domain names.</td>
</tr>
<tr>
<td>parse</td>
<td>logical; should metadata be parsed into a named list (TRUE) or returned as character data?</td>
</tr>
<tr>
<td>name</td>
<td>logical; retrieve name of subdataset? If FALSE, retrieve description</td>
</tr>
<tr>
<td>mask</td>
<td>boolean; stars object with NA mask (0 where NA), or NULL</td>
</tr>
<tr>
<td>use_integer</td>
<td>boolean; if TRUE, raster values are read as (and rounded to) unsigned 32-bit integers values; if FALSE they are read as 32-bit floating points numbers. The former is supposedly faster.</td>
</tr>
<tr>
<td>breaks</td>
<td>numeric vector with break values for contour polygons (or lines)</td>
</tr>
<tr>
<td>use_contours</td>
<td>logical;</td>
</tr>
<tr>
<td>contour_lines</td>
<td>logical;</td>
</tr>
<tr>
<td>connect8</td>
<td>logical; if TRUE use 8 connection algorithm, rather than 4</td>
</tr>
<tr>
<td>sf</td>
<td>object of class sf</td>
</tr>
<tr>
<td>f</td>
<td>character; file name</td>
</tr>
<tr>
<td>pts</td>
<td>points matrix</td>
</tr>
<tr>
<td>bilinear</td>
<td>logical; use bilinear interpolation, rather than nearest neighbor?</td>
</tr>
<tr>
<td>array_name</td>
<td>array name</td>
</tr>
<tr>
<td>offset</td>
<td>offset (pixels)</td>
</tr>
<tr>
<td>count</td>
<td>number of pixels to read</td>
</tr>
<tr>
<td>step</td>
<td>step size (pixels)</td>
</tr>
<tr>
<td>proxy</td>
<td>logical; return proxy object?</td>
</tr>
<tr>
<td>debug</td>
<td>logical; print debug messages?</td>
</tr>
<tr>
<td>dimx</td>
<td>integer named vector with dimensions of object</td>
</tr>
<tr>
<td>cdl</td>
<td>list with variables, each having a named dim attribute</td>
</tr>
<tr>
<td>wkt</td>
<td>character; WKT of crs</td>
</tr>
<tr>
<td>xy</td>
<td>character; names of the spatial x and y dimension</td>
</tr>
<tr>
<td>root_group_options</td>
<td>character; driver specific options regarding the creation of the root group</td>
</tr>
<tr>
<td>as_float</td>
<td>logical; when TRUE write 4-byte floating point numbers, when FALSE write 8-byte doubles.</td>
</tr>
<tr>
<td>nxy</td>
<td>integer vector of length 2</td>
</tr>
<tr>
<td>values</td>
<td>fill value</td>
</tr>
<tr>
<td>crs</td>
<td>object of class crs</td>
</tr>
<tr>
<td>xlim</td>
<td>numeric</td>
</tr>
<tr>
<td>ylim</td>
<td>numeric</td>
</tr>
</tbody>
</table>
Details

These functions are exported for the single purpose of being used by package stars, they are not meant to be used directly and may change or disappear without prior notice or deprecation warnings.

gdal_inv_geotransform returns the inverse geotransform
gdal_crs reads coordinate reference system from GDAL data set
get_metadata gets metadata of a raster layer
gdal_subdatasets returns the subdatasets of a gdal dataset

Value

- object of class crs, see st_crs.
- named list with metadata items

gdal_subdatasets returns a zero-length list if file does not have subdatasets, and else a named list with subdatasets.

Examples

```r
## Not run:
f = system.file("tif/L7_ETMs.tif", package="stars")
f = system.file("nc/avhrr-only-v2.19810901.nc", package = "stars")
gdal_metadata(f)
gdal_metadata(f, NA_character_)
try(gdal_metadata(f, "wrongDomain"))
gdal_metadata(f, c("", "AREA_OR_POINT"))
## End(Not run)
```

**Description**

`gdal_addo` add or remove overviews to/from a raster image

**Usage**

```r
gdal_addo(
  file,
  overviews = c(2, 4, 8, 16),
  method = "NEAREST",
  layers = integer(0),
  options = character(0),
  config_options = character(0),
  clean = FALSE,
  read_only = FALSE
)
```
Arguments

- `file`: character; file name
- `overviews`: integer; overview levels
- `method`: character; method to create overview; one of: nearest, average, rms, gauss, cubic, cubicspline, lanczos, average_mp, average_magphase, mode
- `layers`: integer; layers to create overviews for (default: all)
- `options`: character; dataset opening options
- `config_options`: named character vector with GDAL config options, like `c(option1=value1, option2=value2)`
- `clean`: logical; if TRUE only remove overviews, do not add
- `read_only`: logical; if TRUE, add overviews to another file with extension `.ovr` added to file

Value

TRUE, invisibly, on success

See Also

`gdal_utils` for access to other gdal utilities that have a C API

________________________________________________________________________

`gdal_utils` Native interface to gdal utils

Description

Native interface to gdal utils

Usage

```
gdal_utils(
  util = "info",
  source,
  destination,
  options = character(0),
  quiet = !(util %in% c("info", "gdalinfo", "ogrinfo", "vectorinfo", "mdiminfo")) ||
         ("-multi" %in% options),
  processing = character(0),
  colorfilename = character(0),
  config_options = character(0)
)
```
Arguments

util character; one of info, warp, rasterize, translate, vectortranslate (for ogr2ogr), buildvrt, demprocessing, nearblack, grid, mdiminfo and mdimtranslate (the last two requiring GDAL 3.1), ogrinfo (requiring GDAL 3.7)

source character; name of input layer(s); for warp, buildvrt or mdimtranslate this can be more than one

destination character; name of output layer

options character; options for the utility

quiet logical; if TRUE, suppress printing the output for info and mdiminfo, and suppress printing progress

processing character; processing options for demprocessing

colorfilename character; name of color file for demprocessing (mandatory if processing="color-relief")

config_options named character vector with GDAL config options, like c(option1=value1, option2=value2)

Value

info returns a character vector with the raster metadata; all other utils return (invisibly) a logical indicating success (i.e., TRUE); in case of failure, an error is raised.

See Also

gdal_addo for adding overlays to a raster file; st_layers to query geometry type(s) and crs from layers in a (vector) data source

Examples

if (sf_extSoftVersion()["GDAL"] > "2.1.0") {
  # info utils can be used to list information about a raster dataset. More info: https://gdal.org/programs/ngdalinfo.html
  in_file <- system.file("tif/geomatrix.tif", package = "sf")
  gdal_utils("info", in_file, options = c("-mm", "-proj4"))

  # vectortranslate utils can be used to convert simple features data between file formats. More info: https://gdal.org/programs/ogr2ogr.html
  in_file <- system.file("shape/storms_xyz.shp", package="sf")
  out_file <- paste0(tempfile(), ".gpkg")
  gdal_utils(  
    util = "vectortranslate",
    source = in_file,
    destination = out_file, # output format must be specified for GDAL < 2.3
    options = c("-f", "GPKG")
  )

  # The parameters can be specified as c("name") or c("name", "value"). The # vectortranslate utils can perform also various operations during the # conversion process. For example we can reproject the features during the # translation.
gdal_utils(
    util = "vectortranslate",
    source = in_file,
    destination = out_file,
    options = c(
        "-f", "GPKG", # output file format for GDAL < 2.3
        "-s_srs", "EPSG:4326", # input file SRS
        "-t_srs", "EPSG:2264", # output file SRS
        "-overwrite"
    )
  )
)
st_read(out_file)
  # The parameter s_srs had to be specified because, in this case, the in_file
  # has no associated SRS.
st_read(in_file)
}

---

tools coercion

---

geos_binary_ops

Geometric operations on pairs of simple feature geometry sets

Description

Perform geometric set operations with simple feature geometry collections

Usage

st_intersection(x, y, ...)

## S3 method for class 'sfc'
st_intersection(x, y, ...)

## S3 method for class 'sf'
st_intersection(x, y, ...)

st_difference(x, y, ...)

## S3 method for class 'sfc'
st_difference(x, y, ...)

st_sym_difference(x, y, ...)

st_snap(x, y, tolerance)

Arguments

x  object of class sf, sfc or sfg
y  object of class sf, sfc or sfg
... arguments passed on to s2_options
tolerance values used for `st_snap`; numeric value or object of class `units`; may have tolerance values for each feature in `x`

**Details**


When called with missing `y`, the `sfc` method for `st_intersection` returns all non-empty intersections of the geometries of `x`; an attribute `idx` contains a list-column with the indexes of contributing geometries.

When called with a missing `y`, the `sf` method for `st_intersection` returns an `sf` object with attributes taken from the contributing feature with lowest index; two fields are added: `n.overlaps` with the number of overlapping features in `x`, and a list-column `origins` with indexes of all overlapping features.

When `st_difference` is called with a single argument, overlapping areas are erased from geometries that are indexed at greater numbers in the argument to `x`; geometries that are empty or contained fully inside geometries with higher priority are removed entirely. The `st_difference.sfc` method with a single argument returns an object with an "idx" attribute with the orginal index for returned geometries.

`st_snap` snaps the vertices and segments of a geometry to another geometry's vertices. If `y` contains more than one geometry, its geometries are merged into a collection before snapping to that collection.

(from the GEOS docs:) "A snap distance tolerance is used to control where snapping is performed. Snapping one geometry to another can improve robustness for overlay operations by eliminating nearly-coincident edges (which cause problems during noding and intersection calculation). Too much snapping can result in invalid topology being created, so the number and location of snapped vertices is decided using heuristics to determine when it is safe to snap. This can result in some potential snaps being omitted, however."

**Value**

The intersection, difference or symmetric difference between two sets of geometries. The returned object has the same class as that of the first argument (`x`) with the non-empty geometries resulting from applying the operation to all geometry pairs in `x` and `y`. In case `x` is of class `sf`, the matching attributes of the original object(s) are added. The `sfc` geometry list-column returned carries an attribute `idx`, which is an n-by-2 matrix with every row the index of the corresponding entries of `x` and `y`, respectively.

**Note**

To find whether pairs of simple feature geometries intersect, use the function `st_intersects` instead of `st_intersection`.

When using GEOS and not using `s2` polygons contain their boundary. When using `s2` this is determined by the `model` defaults of `s2_options`, which can be overriden via the `...` argument, e.g. `model = "closed"` to force DE-9IM compliant behaviour of polygons (and reproduce GEOS results).
See Also

- **st_union** for the union of simple features collections; **intersect** and **setdiff** for the base R set operations.

Examples

```r
get.seed(131)
library(sf)
m = rbind(c(0,0), c(1,0), c(1,1), c(0,1), c(0,0))
p = st_polygon(list(m))
n = 100
l = vector("list", n)
for (i in 1:n)
  l[[i]] = p + 10 * runif(2)
s = st_sfc(l)
plot(s, col = sf.colors(categorical = TRUE, alpha = .5))
title("overlapping squares")
d = st_difference(s) # sequential differences: s1, s2-s1, s3-s2-s1, ...
plot(d, col = sf.colors(categorical = TRUE, alpha = .5))
title("non-overlapping differences")
i = st_intersection(s) # all intersections
plot(i, col = sf.colors(categorical = TRUE, alpha = .5))
title("non-overlapping intersections")
s = st_overlaps(s, s)) # includes self-counts!
summary(lengths(st_overlaps(s, s)))
summary(lengths(st_overlaps(d, d)))
summary(lengths(st_overlaps(i, i)))
sf = st_sf(s)
i = st_intersection(s) # all intersections
plot("n.overlaps")
sf = st_sf(s)
n = st_intersection(s) # all intersections
summary(i$n.overlaps - lengths(i$origins))
# A helper function that erases all of y from x:
st_erase = function(x, y) st_difference(x, st_union(st_combine(y)))
poly = st_polygon(list(cbind(c(0, 0), c(1, 1), c(0, 0)))))
lines = st_multilinestring(list(
cbind(c(0, 1), c(0, 1.05)),
cbind(c(0, 1), c(0, -.05)),
cbind(c(1, .95, 1), c(1.05, .5, -.05))
))
snapped = st_snap(poly, lines, tolerance=.1)
plot(snapped, col='red')
plot(poly, border='green', add=TRUE)
plot(lines, lwd=2, col='blue', add=TRUE)
```

---

**geos_binary_pred**

Geometric binary predicates on pairs of simple feature geometry sets

**Description**

Geometric binary predicates on pairs of simple feature geometry sets
Usage

\begin{verbatim}
st_intersects(x, y, sparse = TRUE, ...)  
st_disjoint(x, y = x, sparse = TRUE, prepared = TRUE)  
st_touches(x, y, sparse = TRUE, prepared = TRUE, ...)  
st_crosses(x, y, sparse = TRUE, prepared = TRUE, ...)  
st_within(x, y, sparse = TRUE, prepared = TRUE, ...)  
st_contains(x, y, sparse = TRUE, prepared = TRUE, ..., model = "open")  
st_contains_properly(x, y, sparse = TRUE, prepared = TRUE, ...)  
st_overlaps(x, y, sparse = TRUE, prepared = TRUE, ...)  
st_equals(  
  x,  
  y,  
  sparse = TRUE,  
  prepared = FALSE,  
  ...,  
  retain_unique = FALSE,  
  remove_self = FALSE  
)  
st_covers(x, y, sparse = TRUE, prepared = TRUE, ..., model = "closed")  
st_covered_by(x, y = x, sparse = TRUE, prepared = TRUE, ..., model = "closed")  
st_equals_exact(x, y, par, sparse = TRUE, prepared = FALSE, ...)  
st_is_within_distance(x, y = x, dist, sparse = TRUE, ...)  
\end{verbatim}

Arguments

\begin{verbatim}
x                  object of class sf, sfc or sfg  
y                  object of class sf, sfc or sfg; if missing, x is used  
sparse             logical; should a sparse index list be returned (TRUE) or a dense logical matrix? See below.  
...                 passed on to s2_options  
prepared           logical; prepare geometry for x, before looping over y? See Details.  
model               character; polygon/polyline model; one of "open", "semi-open" or "closed"; see Details.  
retain_unique      logical; if TRUE (and y is missing) return only indexes of points larger than the current index; this can be used to select unique geometries, see examples.  
\end{verbatim}
This argument can be used for all geometry predictates; see also `distinct.sf` to find records where geometries AND attributes are distinct.

**remove_self**

logical; if TRUE (and `y` is missing) return only indexes of geometries different from the current index; this can be used to omit self-intersections; see examples.

This argument can be used for all geometry predictates

**par**

numeric; parameter used for "equals_exact" (margin);

**dist**

distance threshold; geometry indexes with distances smaller or equal to this value are returned; numeric value or units value having distance units.

**Details**

If `prepared` is TRUE, and `x` contains POINT geometries and `y` contains polygons, then the polygon geometries are prepared, rather than the points.

For most predicates, a spatial index is built on argument `x`; see https://r-spatial.org/r/2017/06/22/spatial-index.html. Specifically, `st_intersects`, `st_disjoint`, `stTouches`, `st_crosses`, `st_within`, `stContains`, `stContainsProperly`, `st_overlaps`, `st_equals`, `st_covers` and `st_covered_by` by all build spatial indexes for more efficient geometry calculations. `st_relate`, `st_equals_exact`, and do not; `st_is_within_distance` uses a spatial index for geographic coordinates when `sf_use_s2()` is true.

If `y` is missing, ‘`st_predicate(x, x)`’ is effectively called, and a square matrix is returned with diagonal elements ‘`st_predicate(x[i], x[i])`’.

Sparse geometry binary predicate (sgbp) lists have the following attributes: `region.id` with the row.names of `x` (if any, else 1:n), `ncol` with the number of features in `y`, and `predicate` with the name of the predicate used.

For model, see https://github.com/r-spatial/s2/issues/32

‘`st_contains_properly(A,B)`’ is true if `A` intersects `B`’s interior, but not its edges or exterior; `A` contains `A`, but `A` does not properly contain `A`.

See also `st_relate` and https://en.wikipedia.org/wiki/DE-9IM for a more detailed description of the underlying algorithms.

`st_equals_exact` returns true for two geometries of the same type and their vertices corresponding by index are equal up to a specified tolerance.

**Value**

If `sparse=FALSE`, `st_predicate` (with predicate e.g. "intersects") returns a dense logical matrix with element i,j TRUE when `predicate(x[i], y[j])` (e.g., when geometry of feature i and j intersect); if `sparse=TRUE`, an object of class sgbp with a sparse list representation of the same matrix, with list element i an integer vector with all indices j for which `predicate(x[i], y[j])` is TRUE (and hence a zero-length integer vector if none of them is TRUE). From the dense matrix, one can find out if one or more elements intersect by `apply(mat, 1, any)`, and from the sparse list by `lengths(lst) > 0`, see examples below.

**Note**

For intersection on pairs of simple feature geometries, use the function `st_intersection` instead of `st_intersects`.
Examples

```r
ts <- st_sfc(st_point(c(.5,.5)), st_point(c(1.5, 1.5)), st_point(c(2.5, 2.5)))
pol <- st_polygon(list(rbind(c(0,0), c(2,0), c(2,2), c(0,2), c(0,0))))
(lst <- st_intersects(ts, pol))
(mat <- st_intersects(ts, pol, sparse = FALSE))
# which points fall inside a polygon?
apply(mat, 1, any)
lengths(lst) > 0
# which points fall inside the first polygon?
st_intersects(pol, ts)[[1]]
# remove duplicate geometries:
p1 <- st_point(0:1)
p2 <- st_point(1:2)
p <- st_sfc(a = letters[1:8], geom = st_sfc(p1, p1, p2, p1, p1, p2, p2, p1))
(st.equals(p)
(st.equals(p, remove_self = TRUE)
(u <- st.equals(p, retain_unique = TRUE))
# retain the records with unique geometries:
p[-unlist(u),]
```

---

**geos_combine**

Combine or union feature geometries

**Description**

Combine several feature geometries into one, without unioning or resolving internal boundaries

**Usage**

```r
st_combine(x)
```

```r
st_union(x, y, ..., by_feature = FALSE, is_coverage = FALSE)
```

**Arguments**

- **x**
  - object of class sf, sfc or sfg
- **y**
  - object of class sf, sfc or sfg (optional)
- **...**
  - ignored
- **by_feature**
  - logical; if TRUE, union each feature, if FALSE return a single feature that is the geometric union of the set of features
- **is_coverage**
  - logical; if TRUE, use an optimized algorithm for features that form a polygonal coverage (have no overlaps)
Details

`st_combine` combines geometries without resolving borders, using `c.sfg` (analogous to `c` for ordinary vectors).

If `st_union` is called with a single argument, `x`, (with `y` missing) and `by_feature` is `FALSE` all geometries are unioned together and an `sfg` or single-geometry `sfc` object is returned. If `by_feature` is `TRUE` each feature geometry is unioned. This can for instance be used to resolve internal boundaries after polygons were combined using `st_combine`. If `y` is provided, all elements of `x` and `y` are unioned, pairwise (and `by_feature` is ignored). The former corresponds to `rgeos::gUnaryUnion`, the latter to `rgeos::gUnion`.

Unioning a set of overlapping polygons has the effect of merging the areas (i.e. the same effect as iteratively unioning all individual polygons together). Unioning a set of LineStrings has the effect of fully noding and dissolving the input linework. In this context "fully noded" means that there will be a node or endpoint in the output for every endpoint or line segment crossing in the input. "Dissolved" means that any duplicate (e.g. coincident) line segments or portions of line segments will be reduced to a single line segment in the output. Unioning a set of Points has the effect of merging all identical points (producing a set with no duplicates).

Value

`st_combine` returns a single, combined geometry, with no resolved boundaries; returned geometries may well be invalid.

If `y` is missing, `st_union(x)` returns a single geometry with resolved boundaries, else the geometries for all unioned pairs of `x[i]` and `y[j]`.

See Also

`st_intersection, st_difference, st_sym_difference`

Examples

```r
nc = st_read(system.file("shape/nc.shp", package="sf"))
st_combine(nc)
plot(st_union(nc))
```

---

**Description**

`geos_measures`  
Compute Euclidian or great circle distance between pairs of geometries; compute the area or the length of a set of geometries.
Usage

\texttt{st\_area(x, \ldots)}

\texttt{## S3 method for class 'sfc'}

\texttt{st\_area(x, \ldots)}

\texttt{st\_length(x, \ldots)}

\texttt{st\_distance(}
  \texttt{x,}
  \texttt{y,}
  \texttt{\ldots,}
  \texttt{dist\_fun,}
  \texttt{by\_element = FALSE,}
  \texttt{which = ifelse(isTRUE(st\_is\_longlat(x)), "Great Circle", "Euclidean"),}
  \texttt{par = 0,}
  \texttt{tolerance = 0}
\texttt{)}

Arguments

\texttt{x} \hspace{1cm} \text{object of class sf, sfc or sfg}

\texttt{\ldots} \hspace{1cm} \text{passed on to s2\_distance or s2\_distance\_matrix}

\texttt{y} \hspace{1cm} \text{object of class sf, sfc or sfg, defaults to x}

\texttt{dist\_fun} \hspace{1cm} \text{deprecated}

\texttt{by\_element} \hspace{1cm} \text{logical; if TRUE, return a vector with distance between the first elements of x and y, the second, etc; an error is raised if x and y are not the same length. If FALSE, return the dense matrix with all pairwise distances.}

\texttt{which} \hspace{1cm} \text{character; for Cartesian coordinates only: one of Euclidean, Hausdorff or Frechet; for geodetic coordinates, great circle distances are computed; see details}

\texttt{par} \hspace{1cm} \text{for which equal to Hausdorff or Frechet, optionally use a value between 0 and 1 to densify the geometry}

\texttt{tolerance} \hspace{1cm} \text{ignored if st\_is\_longlat(x) is FALSE; otherwise, if set to a positive value, the first distance smaller than tolerance will be returned, and true distance may be smaller; this may speed up computation. In meters, or a units object convertible to meters.}

Details

great circle distance calculations use by default spherical distances (s2\_distance or s2\_distance\_matrix); if sf\_use\_s2() is FALSE, ellipsoidal distances are computed using st\_geod\_distance which uses function geod\_inverse from GeographicLib (part of PROJ); see Karney, Charles FF, 2013, Algorithms for geodesics, Journal of Geodesy 87(1), 43–55.
Value

If the coordinate reference system of x was set, these functions return values with unit of measurement; see set_units.

st_area returns the area of a geometry, in the coordinate reference system used; in case x is in degrees longitude/latitude, st_geod_area is used for area calculation.

st_length returns the length of a LINESTRING or MULTILINESTRING geometry, using the coordinate reference system. POINT, MULTIPOINT, POLYGON or MULTIPOLYGON geometries return zero.

If by_element is FALSE st_distance returns a dense numeric matrix of dimension length(x) by length(y); otherwise it returns a numeric vector the same length as x and y with an error raised if the lengths of x and y are unequal. Distances involving empty geometries are NA.

See Also

st_dimension, st_cast to convert geometry types

Examples

b0 = st_polygon(list(rbind(c(-1,-1), c(1,-1), c(1,1), c(-1,1), c(-1,-1))))
b1 = b0 + 2
b2 = b0 + c(-0.2, 2)
x = st_sfc(b0, b1, b2)
st_area(x)
line = st_sfc(st_linestring(rbind(c(30,30), c(40,40))), crs = 4326)
st_length(line)

outer = matrix(c(0,0,10,0,10,0,0,10),ncol=2, byrow=TRUE)
hole1 = matrix(c(1,1,1,2,2,1,1,1),ncol=2, byrow=TRUE)
hole2 = matrix(c(5,5,6,6,6,6,5,5),ncol=2, byrow=TRUE)

poly = st_polygon(list(outer, hole1, hole2))
mpoly = st_multipolygon(list(
    list(outer, hole1, hole2),
    list(outer + 12, hole1 + 12)
))
st_length(st_sfc(poly, mpoly))
p = st_sfc(st_point(c(0,0)), st_point(c(0,1)), st_point(c(0,2)))
st_distance(p, p)
st_distance(p, p, by_element = TRUE)
Usage

\[
\text{st\_dimension}(x, \text{NA\_if\_empty} = \text{TRUE})
\]

\[
\text{st\_is\_simple}(x)
\]

\[
\text{st\_is\_empty}(x)
\]

Arguments

- **x**: object of class sf, sfc or sfg
- **NA\_if\_empty**: logical; if TRUE, return NA for empty geometries

Value

- \text{st\_dimension} returns a numeric vector with 0 for points, 1 for lines, 2 for surfaces, and, if NA\_if\_empty is TRUE, NA for empty geometries.
- \text{st\_is\_simple} returns a logical vector, indicating for each geometry whether it is simple (e.g., not self-intersecting)
- \text{st\_is\_empty} returns for each geometry whether it is empty

Examples

```r
x = \text{st\_sfc}(
\text{st\_point}(0:1),
\text{st\_linestring}(\text{cbind}(c(0,0),c(1,1)))),
\text{st\_polygon}([[\text{cbind}(c(0,0),c(1,0),c(0,1),c(0,0))]]),
\text{st\_multipoint}(),
\text{st\_linestring}(),
\text{st\_geometrycollection}())
\text{st\_dimension}(x)
\text{st\_dimension}(x, \text{FALSE})
ls = \text{st\_linestring}(\text{cbind}(c(0,0),c(1,0),c(1,1),c(0,1))))
\text{st\_is\_simple}(\text{st\_sfc}(ls, \text{st\_point}(c(0,0))))
ls = \text{st\_linestring}(\text{cbind}(c(0,0),c(1,1),c(1,0),c(0,1))))
\text{st\_is\_empty}(\text{st\_sfc}(ls, \text{st\_point}(), \text{st\_linestring}()))
```

---

**Description**

Geometric unary operations on simple feature geometry sets. These are all generics, with methods for sfg, sfc and sf objects, returning an object of the same class. All operations work on a per-feature basis, ignoring all other features.
Usage

```r
st_buffer(
  x,
  dist,
  nQuadSegs = 30,
  endCapStyle = "ROUND",
  joinStyle = "ROUND",
  mitreLimit = 1,
  singleSide = FALSE,
  ...
)
```

```r
st_boundary(x)
```

```r
st_convex_hull(x)
```

```r
st_concave_hull(x, ratio, ..., allow_holes)
```

```r
st_simplify(x, preserveTopology, dTolerance = 0)
```

```r
st_simplify(x, dTolerance = 0, bOnlyEdges = FALSE)
```

```r
st_triangulate_constrained(x)
```

```r
st_triangulate(x, dTolerance = 0, bOnlyEdges = FALSE)
```

```r
st_inscribed_circle(x, dTolerance, ...)
```

```r
st_minimum_rotated_rectangle(x, ...)
```

```r
st_voronoi(x, envelope, dTolerance = 0, bOnlyEdges = FALSE)
```

```r
st_polygonize(x)
```

```r
st_line_merge(x)
```

```r
st_centroid(x, ..., of_largest_polygon = FALSE)
```

```r
st_point_on_surface(x)
```

```r
st_reverse(x)
```

```r
st_node(x)
```

```r
st_segmentize(x, dfMaxLength, ...)
```

Arguments

- **x**: object of class `sfg`, `sfc` or `sf`
- **dist**: numeric; buffer distance for all, or for each of the elements in `x`; in case
is a units object, it should be convertible to `arc_degree` if `x` has geographic
coordinates, and to `st_crs(x)$units` otherwise

- `nQuadSegs` - integer; number of segments per quadrant (fourth of a circle), for all or per-
  feature; see details
- `endCapStyle` - character; style of line ends, one of 'ROUND', 'FLAT', 'SQUARE'; see details
- `joinStyle` - character; style of line joins, one of 'ROUND', 'MITRE', 'BEVEL'; see details
- `mitreLimit` - numeric; limit of extension for a join if `joinStyle` 'MITRE' is used (default
  1.0, minimum 0.0); see details
- `singleSide` - logical; if TRUE, single-sided buffers are returned for linear geometries, in
  which case negative `dist` values give buffers on the right-hand side, positive on the
  left; see details
- `...` - ignored
- `ratio` - numeric; fraction convex: 1 returns the convex hulls, 0 maximally concave hulls
- `allow_holes` - logical; if TRUE, the resulting concave hull may have holes
- `preserveTopology` - logical; carry out topology preserving simplification? May be specified for each,
  or for all feature geometries. Note that topology is preserved only for single feature
  geometries, not for sets of them. If not specified (i.e. the default), then it
  is internally set equal to FALSE when the input data is specified with
  projected coordinates or `sf_use_s2()` returns FALSE. Ignored in all the other cases
  (with a warning when set equal to FALSE) since the function implicitly calls
  `s2::s2_simplify` which always preserve topological relationships (per single
  feature).
- `dTolerance` - numeric; tolerance parameter, specified for all or for each feature geometry. If you
  run `st_simplify`, the input data is specified with long-lat coordinates and
  `sf_use_s2()` returns TRUE, then the value of `dTolerance` must be specified in
  meters.
- `bOnlyEdges` - logical; if TRUE, return lines, else return polygons
- `envelope` - object of class `sfc` or `sfg` containing a `POLYGON` with the envelope for a voronoi
diagram; this only takes effect when it is larger than the default envelope, chosen
when `envelope` is an empty polygon
- `of_largest_polygon` - logical; for `st_centroid`: if TRUE, return centroid of the largest (sub)polygon
  of a `MULTIPOLYGON` rather than of the whole `MULTIPOLYGON`
- `dfMaxLength` - maximum length of a line segment. If `x` has geographical coordinates (long/lat),
  `dfMaxLength` is either a numeric expressed in meter, or an object of class `units` with
  length units rad or degree; segmentation in the long/lat case takes place
  along the great circle, using `st_geod_segmentize`.

**Details**

`st_buffer` computes a buffer around this geometry/each geometry. If any of `endCapStyle`, `joinStyle`,
or `mitreLimit` are set to non-default values ('ROUND', 'ROUND', 1.0 respectively) then the under-
lying 'buffer with style' GEOS function is used. If a negative buffer returns empty polygons
instead of shrinking, set `st_use_s2()` to FALSE. See `postgis.net/docs/ST_Buffer.html` for details.
nQuadSegs, endCapStyle, joinStyle, mitreLimit and singleSide only work when the GEOS back-end is used: for projected coordinates or when sf_use_s2() is set to FALSE.

st_boundary returns the boundary of a geometry

st_convex_hull creates the convex hull of a set of points

st_concave_hull creates the concave hull of a geometry

st_simplify simplifies lines by removing vertices.

st_triangulate triangulates set of points (not constrained). st_triangulate requires GEOS version 3.4 or above

st_triangulate_constrained returns the constrained delaunay triangulation of polygons; requires GEOS version 3.10 or above

st_inscribed_circle returns the maximum inscribed circle for polygon geometries. For st_inscribed_circle, if nQuadSegs is 0 a 2-point LINESTRING is returned with the center point and a boundary point of every circle, otherwise a circle (buffer) is returned where nQuadSegs controls the number of points per quadrant to approximate the circle. st_inscribed_circle requires GEOS version 3.9 or above

st_minimum_rotated_rectangle returns the minimum rotated rectangular POLYGON which encloses the input geometry. The rectangle has width equal to the minimum diameter, and a longer length. If the convex hull of the input is degenerate (a line or point) a linestring or point is returned.

st_voronoi creates voronoi tessellation. st_voronoi requires GEOS version 3.5 or above

st_polygonize creates polygon from lines that form a closed ring. In case of st_polygonize, x must be an object of class LINESTRING or MULTILINESTRING, or an sfc geometry list-column object containing these

st_line_merge merges lines. In case of st_line_merge, x must be an object of class MULTILINESTRING, or an sfc geometry list-column object containing these

st_centroid gives the centroid of a geometry

st_point_on_surface returns a point guaranteed to be on the (multi)surface.

st_reverse reverses the nodes in a line

st_node adds nodes to linear geometries at intersections without a node, and only works on individual linear geometries

st_segmentize adds points to straight lines

Value

an object of the same class of x, with manipulated geometry.

See Also

chull for a more efficient algorithm for calculating the convex hull

Examples

```r
## st_buffer, style options (taken from rgeos gBuffer)
l1 = st_as_sfc("LINESTRING(0 0,1 5,4 5,5 2,8 2,9 4,4 6.5)")
op = par(mfrow=c(2,3))
```
plot(st_buffer(l1, dist = 1, endCapStyle="ROUND"), reset = FALSE, main = "endCapStyle: ROUND")
plot(l1,col='blue',add=TRUE)
plot(st_buffer(l1, dist = 1, endCapStyle="FLAT"), reset = FALSE, main = "endCapStyle: FLAT")
plot(l1,col='blue',add=TRUE)
plot(st_buffer(l1, dist = 1, endCapStyle="SQUARE"), reset = FALSE, main = "endCapStyle: SQUARE")
plot(l1,col='blue',add=TRUE)
plot(st_buffer(l1, dist = 1, nQuadSegs=1), reset = FALSE, main = "nQuadSegs: 1")
plot(l1,col='blue',add=TRUE)
plot(st_buffer(l1, dist = 1, nQuadSegs=2), reset = FALSE, main = "nQuadSegs: 2")
plot(l1,col='blue',add=TRUE)
plot(st_buffer(l1, dist = 1, nQuadSegs= 5), reset = FALSE, main = "nQuadSegs: 5")
plot(l1,col='blue',add=TRUE)
par(op)

l2 = st_as_sfc("LINESTRING(0 0,1 5,3 2)")
op = par(mfrow = c(2, 3))
plot(st_buffer(l2, dist = 1, joinStyle="ROUND"), reset = FALSE, main = "joinStyle: ROUND")
plot(l2, col = 'blue', add = TRUE)
plot(st_buffer(l2, dist = 1, joinStyle="MITRE"), reset = FALSE, main = "joinStyle: MITRE")
plot(l2, col = 'blue', add = TRUE)
plot(st_buffer(l2, dist = 1, joinStyle="BEVEL"), reset = FALSE, main = "joinStyle: BEVEL")
plot(l2, col = 'blue', add = TRUE)
plot(st_buffer(l2, dist = 1, joinStyle="MITRE", mitreLimit=0.5), reset = FALSE,
main = "mitreLimit: 0.5")
plot(l2, col = 'blue', add = TRUE)
plot(st_buffer(l2, dist = 1, joinStyle="MITRE", mitreLimit=1), reset = FALSE,
main = "mitreLimit: 1")
plot(l2, col = 'blue', add = TRUE)
plot(st_buffer(l2, dist = 1, joinStyle="MITRE", mitreLimit=3), reset = FALSE,
main = "mitreLimit: 3")
plot(l2, col = 'blue', add = TRUE)
par(op)
nc = st_read(system.file("shape/nc.shp", package="sf"))
nc_g = st_geometry(nc)
plot(st_convex_hull(nc_g))
plot(nc_g, border = grey(.5), add = TRUE)
set.seed(131)
if (compareVersion(sf_extSoftVersion()[["GEOS"]], "3.11.0") > -1) {
  pts = cbind(runif(100), runif(100))
m = st_multipoint(pts)
co = st:::st_concave_hull(m, 0.3)
coh = sf:::st_concave_hull(m, 0.3, allow_holes = TRUE)
plot(co, col = 'grey')
plot(coh, add = TRUE, border = 'red')
plot(m, add = TRUE)
}

# st_simplify examples:
op = par(mfrow = c(2, 3), mar = rep(0, 4))
plot(nc_g[1])
plot(st_simplify(nc_g[1], dTolerance = 1e3)) # 1000m
plot(st_simplify(nc_g[1], dTolerance = 5e3)) # 5000m
nc_g_planar = st_transform(nc_g, 2264) # planar coordinates, US foot
plot(nc_g_planar[1])
plot(st_simplify(nc_g_planar[1], dTolerance = 1e3)) # 1000 foot
plot(st_simplify(nc_g_planar[1], dTolerance = 5e3)) # 5000 foot
par(op)
if (compareVersion(sf_extSoftVersion()["GEOS"], "3.10.0") > -1) {
  pts = rbind(c(0,0), c(1,0), c(1,1), c(.5,.5), c(0,1), c(0,0))
pol = st_polygon(list(pts))
co = st_triangulate_constrained(pol)
tr = st_triangulate(pol)
plot(pol, col = NA, border = 'grey', lwd = 15)
plot(tr, border = 'green', col = NA, lwd = 5, add = TRUE)
plot(co, border = 'red', col = 'NA', add = TRUE)
}
if (compareVersion(sf_extSoftVersion()["GEOS"], "3.9.0") > -1) {
  nc_t = st_transform(nc, 'EPSG:2264')
x = st_inscribed_circle(st_geometry(nc_t))
plot(st_geometry(nc_t), asp = 1, col = grey(.9))
plot(x, add = TRUE, col = '#FF9999')
}
set.seed(1)
x = st_multipoint(matrix(runif(10),,2))
box = st_polygon(list(rbind(c(0,0),c(1,0),c(1,1),c(0,1),c(0,0))))
if (compareVersion(sf_extSoftVersion()["GEOS"], "3.5.0") > -1) {
v = st_sfc(st_voronoi(x, st_sfc(box)))
plot(v, col = 0, border = 1, axes = TRUE)
plot(box, add = TRUE, col = 0, border = 1) # a larger box is returned, as documented
plot(v, add = TRUE, col = 'red', cex=2, pch=16)
plot(st_intersection(st_cast(v), box)) # clip to smaller box
plot(x, add = TRUE, col = 'red', cex=2, pch=16)
# matching Voronoi polygons to data points:
# https://github.com/r-spatial/sf/issues/1030
# generate 50 random unif points:
n = 100
pts = st_as_sf(data.frame(matrix(runif(n), , 2), id = 1:(n/2)), coords = c("X1", "X2"))
# compute Voronoi polygons:
pols = st_collection_extract(st_voronoi(do.call(c, st_geometry(pts))))
# match them to points:
plot(pts[pols[unlist(st_intersects(pts, pols))]]
plot(pts["id"], pch = 16) # ID is color
plot(st_set_geometry(pts, "pols"["id"], xlab = c(0,1), ylab = c(0,1), reset = FALSE)
layout(matrix(1)) # reset plot layout
}
mls = st_multilinestring(list(matrix(c(0,0,0,1,1,1,0,0),,2,byrow=TRUE)))
st_polygonize(st_sfc(mls))
mls = st_multilinestring(list(rbind(c(0,0), c(1,1)), rbind(c(2,0), c(1,1))))
st_line_merge(st_sfc(mls))
plot(nc_g, axes = TRUE)
plot(st_centroid(nc_g), add = TRUE, pch = 3, col = 'red')
mp = st_combine(st_buffer(st_sfc(lapply(1:3, function(x) st_point(c(x,x)))), 0.2 * 1:3))
plot(mp)
plot(st_centroid(mp), add = TRUE, col = 'red') # centroid of combined geometry
plot(st_centroid(mp, of_largest_polygon = TRUE), add = TRUE, col = 'blue', pch = 3)
plot(nc_g, axes = TRUE)
plot(st_point_on_surface(nc_g), add = TRUE, pch = 3, col = 'red')
if (compareVersion(sf_extSoftVersion()["GEOS"], "3.7.0") > -1) {
  st_reverse(st_linestring(rbind(c(1,1), c(2,2), c(3,3))))
}
(l = st_linestring(rbind(c(0,0), c(1,1), c(0,1), c(1,0), c(0,0))))
st_polygonize(st_node(l))
st_node(st_multilinestring(list(rbind(c(0,0), c(1,1), c(0,1), c(1,0), c(0,0)))))
sf = st_sf(a=1, geom=st_sfc(st_linestring(rbind(c(0,0),c(1,1))))), crs = 4326)
if (require(lwgeom, quietly = TRUE)) {
  seg = st_segmentize(sf, units::set_units(100, km))
  seg = st_segmentize(sf, units::set_units(0.01, rad))
nrow(seg$geom[[1]])
}

```r
internal

Internal functions
```

**Description**

Internal functions

**Usage**

`.stop_geos(msg)`

**Arguments**

- `msg` error message

```r
interpolate_aw

Areal-weighted interpolation of polygon data
```

**Description**

Areal-weighted interpolation of polygon data

**Usage**

`st_interpolate_aw(x, to, extensive, ...)`

```r
## S3 method for class 'sf'
st_interpolate_aw(x, to, extensive, ..., keep_NA = FALSE)
```
**is_driver_available**

**Arguments**

- **x**: object of class `sf`, for which we want to aggregate attributes
- **to**: object of class `sf` or `sfc`, with the target geometries
- **extensive**: logical; if `TRUE`, the attribute variables are assumed to be spatially extensive (like population) and the sum is preserved, otherwise, spatially intensive (like population density) and the mean is preserved.
- **...**: ignored
- **keep_NA**: logical; if `TRUE`, return all features in to, if `FALSE` return only those with non-NA values (but with `row.names` the index corresponding to the feature in to)

**Examples**

```r
nc = st_read(system.file("shape/nc.shp", package="sf"))
g = st_make_grid(nc, n = c(10, 5))
a1 = st_interpolate_aw(nc["BIR74"], g, extensive = FALSE)
sum(a1$BIR74) / sum(nc$BIR74) # not close to one: property is assumed spatially intensive
a2 = st_interpolate_aw(nc["BIR74"], g, extensive = TRUE)
# verify mass preservation (pycnophylactic) property:
sum(a2$BIR74) / sum(nc$BIR74)
a1$intensive = a1$BIR74
a1$extensive = a2$BIR74
plot(a1[c("intensive", "extensive")], key.pos = 4)
```

---

**is_driver_available**  
*Check if driver is available*

**Description**

Search through the driver table if driver is listed

**Usage**

```r
is_driver_available(drv, drivers = st_drivers())
```

**Arguments**

- **drv**: character. Name of driver
- **drivers**: data.frame. Table containing driver names and support. Default is from `st_drivers`
is_driver_can  
*Check if a driver can perform an action*

**Description**

Search through the driver table to match a driver name with an action (e.g. "write") and check if the action is supported.

**Usage**

```r
is_driver_can(drv, drivers = st_drivers(), operation = "write")
```

**Arguments**

- **drv**
  - character. Name of driver
- **drivers**
  - data.frame. Table containing driver names and support. Default is from `st_drivers`
- **operation**
  - character. What action to check

---

is_geometry_column  
*Check if the columns could be of a coercable type for sf*

**Description**

Check if the columns could be of a coercable type for `sf`

**Usage**

```r
is_geometry_column(con, x, classes = "")
```

**Arguments**

- **con**
  - database connection
- **x**
  - inherits data.frame
- **classes**
  - classes inherited
merge.sf

merge sf and data.frame object

Description
merge sf and data.frame object

Usage

## S3 method for class 'sf'
merge(x, y, ...)

Arguments

x object of class sf
y object of class data.frame
... arguments passed on to merge.data.frame

Examples

a = data.frame(a = 1:3, b = 5:7)
st_geometry(a) = st_sfc(st_point(c(0,0)), st_point(c(1,1)), st_point(c(2,2)))
b = data.frame(x = c("a", "b", "c"), b = c(2,5,6))
merge(a, b)
merge(a, b, all = TRUE)

nc

North Carolina SIDS data

Description
Sudden Infant Death Syndrome (SIDS) sample data for North Carolina counties, two time periods (1974-78 and 1979-84). The details of the columns can be found on the seealso URL, spdep package’s vignette. Please note that, though this is basically the same as nc.sids dataset in spData package, nc only contains a subset of variables. The differences are also discussed on the vignette.

See Also

https://r-spatial.github.io/spdep/articles/sids.html
**Ops**

*S3 Ops Group Generic Functions for simple feature geometries*

**Description**

S3 Ops Group Generic Functions for simple feature geometries

**Usage**

```r
## S3 method for class 'sfg'
Ops(e1, e2)

## S3 method for class 'sfc'
Ops(e1, e2)
```

**Arguments**

- `e1` object of class `sfg` or `sfc`
- `e2` numeric, or object of class `sfg`; in case `e1` is of class `sfc` also an object of class `sfc` is allowed

**Details**

In case `e2` is numeric, `+`, `-`, `*`, `/,

If `e1` is of class `sfc`, and `e2` is a length 2 numeric, then it is considered a two-dimensional point (and if needed repeated as such) only for operations `+` and `-`, in other cases the individual numbers are repeated; see commented examples.

It has been reported (https://github.com/r-spatial/sf/issues/2067) that certain ATLAS versions result in invalid polygons, where the final point in a ring is no longer equal to the first point. In that case, setting the precisions with `st_set_precision` may help.

**Value**

object of class `sfg`

**Examples**

```r
st_point(c(1,2,3)) + 4
st_point(c(1,2,3)) * 3 + 4
m = matrix(0, 2, 2)
diag(m) = c(1, 3)
# affine:
st_point(c(1,2)) * m + c(2,5)
# world in 0-360 range:
if (require(maps, quietly = TRUE)) {
  w = st_as_sf(map('world', plot = FALSE, fill = TRUE))
  w2 = (st_geometry(w) + c(360,90)) %% c(360) - c(0,90)
```
```r
w3 = st_wrap_dateline(st_set_crs(w2 - c(180,0), 4326)) + c(180,0)
plot(st_set_crs(w3, 4326), axes = TRUE)

(mp <- st_point(c(1,2)) + st_point(c(3,4))) # MULTIPOINT (1 2, 3 4)
mp - st_point(c(3,4)) # POINT (1 2)
opar = par(mfrow = c(2,2), mar = c(0, 0, 1, 0))
a = st_buffer(st_point(c(0,0)), 2)
b = a + c(2, 0)
p = function(m) { plot(c(a,b)); plot(eval(parse(text=m)), col=grey(.9), add = TRUE); title(m) }
lapply(c(quotesingle.Var a | b, 'a / b', 'a & b', 'a %% b'), p)
par(opar)

sfc = st_sfc(st_point(0:1), st_point(2:3))
sfc + c(2,3) # added to EACH geometry
sfc * c(2,3) # first geometry multiplied by 2, second by 3
nc = st_transform(st_read(system.file("gpkg/nc.gpkg", package="sf"), 32119) # nc state plane, m
b = st_buffer(st_centroid(st_union(nc)), units::set_units(50, km)) # shoot a hole in nc:
plot(st_geometry(nc) / b, col = grey(.9))
```

---

**Description**

plot one or more attributes of an sf object on a map

**Plot sf object**

**Usage**

```r
# S3 method for class 'sf'
plot(
x,
y,
..., 
main,
pal = NULL,
nbreaks = 10,
breaks = "pretty",
max.plot = if (is.null(n <- getOption("sf_max.plot"))) 9 else n,
key.pos = get_key_pos(x, ...),
key.length = 0.618,
key.width = lcm(1.8),
reset = TRUE,
logz = FALSE,
extent = x,
xlim = st_bbox(extent)[c(1, 3)],
ylim = st_bbox(extent)[c(2, 4)]
)
```

get_key_pos(x, ...)
## S3 method for class 'sfc_POINT'
plot(
  x,
  y,
  ..., 
  pch = 1,
  cex = 1,
  col = 1,
  bg = 0,
  lwd = 1,
  lty = 1,
  type = "p",
  add = FALSE
)

## S3 method for class 'sfc_MULTIPOINT'
plot(
  x,
  y,
  ..., 
  pch = 1,
  cex = 1,
  col = 1,
  bg = 0,
  lwd = 1,
  lty = 1,
  type = "p",
  add = FALSE
)

## S3 method for class 'sfc_LINESTRING'
plot(x, y, ..., lty = 1, lwd = 1, col = 1, pch = 1, type = "l", add = FALSE)

## S3 method for class 'sfc_CIRCULARSTRING'
plot(x, y, ...)

## S3 method for class 'sfc_MULTILINESTRING'
plot(x, y, ..., lty = 1, lwd = 1, col = 1, pch = 1, type = "l", add = FALSE)

## S3 method for class 'sfc_POLYGON'
plot(
  x,
  y,
  ..., 
  lty = 1,
  lwd = 1,
  col = NA,
```r
plot(cex = 1,
     pch = NA,
     border = 1,
     add = FALSE,
     rule = "evenodd",
     xpd = par("xpd")
)

## S3 method for class 'sfc_MULTIPOLYGON'
plot(
     x,
     y,
     ..., 
     lty = 1,
     lwd = 1,
     col = NA,
     border = 1,
     add = FALSE,
     rule = "evenodd",
     xpd = par("xpd")
)

## S3 method for class 'sfc_GEOMETRYCOLLECTION'
plot(
     x,
     y,
     ..., 
     pch = 1,
     cex = 1,
     bg = 0,
     lty = 1,
     lwd = 1,
     col = 1,
     border = 1,
     add = FALSE
)

## S3 method for class 'sfc_GEOMETRY'
plot(
     x,
     y,
     ..., 
     pch = 1,
     cex = 1,
     bg = 0,
     lty = 1,
     lwd = 1,
     col = ifelse(st_dimension(x) == 2, NA, 1),
```
border = 1,
add = FALSE
)

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

plot_sf(
  x,
  xlim = NULL,
  ylim = NULL,
  asp = NA,
  axes = FALSE,
  bgc = par("bg"),
  ..., 
  xaxs, 
  yaxs, 
  lab,
  setParUsrBB = FALSE,
  bgMap = NULL,
  expandBB = c(0, 0, 0, 0),
  graticule = NA_crs_,
  col_graticule = "grey",
  border,
  extent = x
)

sf.colors(n = 10, cutoff.tails = c(0.35, 0.2), alpha = 1, categorical = FALSE)

Arguments

x object of class sf
y ignored
... further specifications, see plot.sf and plot and details.
main title for plot (NULL to remove)
pal palette function, similar to rainbow, or palette values; if omitted, sf.colors is used
nbbreaks number of colors breaks (ignored for factor or character variables)
breaks either a numeric vector with the actual breaks, or a name of a method accepted by the style argument of classIntervals
max.plot integer; lower boundary to maximum number of attributes to plot; the default value (9) can be overriden by setting the global option sf_max.plot, e.g. options(sf_max.plot=2)
key.pos integer; side to plot a color key: 1 bottom, 2 left, 3 top, 4 right; set to NULL to omit key completely, 0 to only not plot the key, or -1 to select automatically. If multiple columns are plotted in a single function call by default no key is plotted and every submap is stretched individually; if a key is requested (and col is
missing) all maps are colored according to a single key. Auto select depends on plot size, map aspect, and, if set, parameter `asp`.

- **key.length**: amount of space reserved for the key along its axis, length of the scale bar
- **key.width**: amount of space reserved for the key (incl. labels), thickness/width of the scale bar
- **reset**: logical; if FALSE, keep the plot in a mode that allows adding further map elements; if TRUE restore original mode after plotting sf objects with attributes; see details.
- **logz**: logical; if TRUE, use log10-scale for the attribute variable. In that case, breaks and at need to be given as log10-values; see examples.
- **extent**: object with an `st_bbox` method to define plot extent; defaults to `x`
- **xlim** and **ylim**: see `plot.window`
- **pch**: plotting symbol
- **cex**: symbol size
- **col**: color for plotting features; if `length(col)` does not equal 1 or `nrow(x)`, a warning is emitted that colors will be recycled. Specifying `col` suppresses plotting the legend key.
- **bg**: symbol background color
- **lwd**: line width
- **lty**: line type
- **type**: plot type: 'p' for points, 'l' for lines, 'b' for both
- **add**: logical; add to current plot? Note that when using `add=TRUE`, you may have to set `reset=FALSE` in the first plot command.
- **border**: color of polygon border(s); using NA hides them
- **rule**: see `polypath`; for winding, exterior ring direction should be opposite that of the holes; with `evenodd`, plotting is robust against misspecified ring directions
- **xpd**: see `par`; sets polygon clipping strategy; only implemented for POLYGON and MULTIPOLYGON
- **asp**: see below, and see `par`
- **axes**: logical; should axes be plotted? (default FALSE)
- **bgc**: background color
- **xaxs** and **yaxs**: see `par`
- **lab**: see `par`
- **setParUsrBB**: default FALSE; set the `par` “usr” bounding box; see below
- **bgMap**: object of class ggmap, or returned by function `RgoogleMaps::GetMap`
- **expandBB**: numeric; fractional values to expand the bounding box with, in each direction (bottom, left, top, right)
graticule logical, or object of class crs (e.g., st_crs(4326) for a WGS84 graticule), or object created by st_graticule; TRUE will give the WGS84 graticule or object returned by st_graticule

col_graticule color to used for the graticule (if present)
n integer; number of colors
cutoff.tails numeric, in [0,0.5] start and end values
alpha numeric, in [0,1], transparency
categorical logical; do we want colors for a categorical variable? (see details)

Details
plot.sf maximally plots max.plot maps with colors following from attribute columns, one map per attribute. It uses sf.colors for default colors. For more control over placement of individual maps, set parameter mfrom with par prior to plotting, and plot single maps one by one; note that this only works in combination with setting parameters key.pos=NULL (no legend) and reset=FALSE.

plot.sfc plots the geometry, additional parameters can be passed on to control color, lines or symbols.

When setting reset to FALSE, the original device parameters are lost, and the device must be reset using dev.off() in order to reset it.

parameter at can be set to specify where labels are placed along the key; see examples.

The features are plotted in the order as they appear in the sf object. See examples for when a different plotting order is wanted.

plot_sf sets up the plotting area, axes, graticule, or webmap background; it is called by all plot methods before anything is drawn.

The argument setParUsrBB may be used to pass the logical value TRUE to functions within plot.Spatial. When set to TRUE, par("usr") will be overwritten with c(xlim, ylim), which defaults to the bounding box of the spatial object. This is only needed in the particular context of graphic output to a specified device with given width and height, to be matched to the spatial object, when using par("xaxs") and par("yaxs") in addition to par(mar=c(0,0,0,0)).

The default aspect for map plots is 1: if however data are not projected (coordinates are long/lat), the aspect is by default set to 1/cos(My * pi/180) with My the y coordinate of the middle of the map (the mean of ylim, which defaults to the y range of bounding box). This implies an Equirectangular projection.

non-categorical colors from sf.colors were taken from bpy.colors, with modified cutoff.tails defaults If categorical is TRUE, default colors are from https://colorbrewer2.org/ (if n < 9, Set2, else Set3).

Examples
nc = st_read(system.file("gpkg/nc.gpkg", package="sf"), quiet = TRUE)
# plot single attribute, auto-legend:
plot(nc["SID74"])
# plot multiple:
plot(nc[,c("SID74", "SID79")]) # better use ggplot2::geom_sf to facet and get a single legend!
# adding to a plot of an sf object only works when using reset=FALSE in the first plot:
plot(nc[, "SID74"], reset = FALSE)
plot(st_centroid(st_geometry(nc)), add = TRUE)
# log10 z-scale:
plot(nc[, "SID74"], logz = TRUE, breaks = c(0, .5, 1, 1.5, 2), at = c(0, .5, 1, 1.5, 2))
# and we need to reset the plotting device after that, e.g. by
layout(1)
# when plotting only geometries, the reset=FALSE is not needed:
plot(st_geometry(nc))
plot(st_geometry(nc)[1], col = 'red', add = TRUE)
# add a custom legend to an arbitrary plot:
layout(matrix(1:2, ncol = 2), widths = c(1, lcm(2)))
plot(1)
.image_scale(1:10, col = sf.colors(9), key.length = lcm(8), key.pos = 4, at = 1:10)
# manipulate plotting order, plot largest polygons first:
p = st_polygon(list(rbind(c(0, 0), c(1, 0), c(1, 1), c(0, 1), c(0, 0))))
x = st_sf(a=1:4, st_sfc(p, p * 2, p * 3, p * 4)) # plot(x, col=2:5) only shows the largest polygon!
plot(x[order(st_area(x), decreasing = TRUE),], col = 2:5) # plot largest polygons first
sf.colors(10)

---

**prefix_map**

*Map prefix to driver*

**Description**

Map prefix to driver

**Usage**

prefix_map

**Format**

An object of class list of length 10.

---

**proj_tools**

*Manage PROJ settings*

**Description**

Manage PROJ search path and network settings
Usage

sf_proj_search_paths(paths = character(0))

sf_proj_network(enable = FALSE, url = character(0))

sf_proj_pipelines(
  source_crs,
  target_crs,
  authority = character(0),
  AOI = numeric(0),
  Use = "NONE",
  grid_availability = "USED",
  desired_accuracy = -1,
  strict_containment = FALSE,
  axis_order_authority_compliant = st_axis_order()
)

Arguments

paths the search path to be set; omit if no paths need to be set
enable logical; set this to enable (TRUE) or disable (FALSE) the proj network search facility
url character; use this to specify and override the default proj network CDN
source_crs object of class ‘crs’ or character
target_crs object of class ‘crs’ or character
authority character; constrain output pipelines to those of authority
AOI length four numeric; desired area of interest for the resulting coordinate transformations (west, south, east, north, in degrees). For an area of interest crossing the anti-meridian, west will be greater than east.
Use one of "NONE", "BOTH", "INTERSECTION", "SMALLEST", indicating how AOI’s of source_crs and target_crs are being used
grid_availability character; one of "USED" (Grid availability is only used for sorting results. Operations where some grids are missing will be sorted last), "DISCARD" (Completely discard an operation if a required grid is missing), "IGNORED" (Ignore grid availability at all. Results will be presented as if all grids were available.), or "AVAILABLE" (Results will be presented as if grids known to PROJ (that is registered in the grid_alternatives table of its database) were available. Used typically when networking is enabled.)
desired_accuracy numeric; only return pipelines with at least this accuracy
strict_containment logical; default FALSE; permit partial matching of the area of interest; if TRUE strictly contain the area of interest. The area of interest is either as given in AOI, or as implied by the source/target coordinate reference systems
axis_order_authority_compliant
  logical; if FALSE always choose ‘x’ or longitude for the first axis; if TRUE, follow the axis orders given by the coordinate reference systems when constructing the for the first axis; if FALSE, follow the axis orders given by

Value
  ‘sf_proj_search_paths()’ returns the search path (possibly after setting it)
  ‘sf_proj_network’ when called without arguments returns a logical indicating whether network search of datum grids is enabled, when called with arguments it returns a character vector with the URL of the CDN used (or specified with ‘url’).
  ‘sf_proj_pipelines’ returns a table with candidate coordinate transformation pipelines along with their accuracy; ‘NA’ accuracy indicates ballpark accuracy.

---

**rawToHex**

*Convert raw vector(s) into hexadecimal character string(s)*

**Description**

Convert raw vector(s) into hexadecimal character string(s)

**Usage**

`rawToHex(x)`

**Arguments**

*x*
  raw vector, or list with raw vectors

---

**s2**

*functions for spherical geometry, using s2 package*

**Description**

functions for spherical geometry, using the s2 package based on the google s2geometry.io library

**Usage**

`sfc_use_s2(use_s2)`

`st_as_s2(x, ...)`

## S3 method for class 'sf'
`st_as_s2(x, ...)`

## S3 method for class 'sfc'
`st_as_s2(x, ..., oriented = getOption("s2_oriented", FALSE), rebuild = FALSE)`
Arguments

- **use_s2**
  - logical; if TRUE, use the s2 spherical geometry package for geographical coordinate operations
- **x**
  - object of class sf, sfc or sfg
- **...**
  - passed on
- **oriented**
  - logical; if FALSE, polygons that cover more than half of the globe are inverted; if TRUE, no reversal takes place and it is assumed that the inside of the polygon is to the left of the polygon’s path.
- **rebuild**
  - logical; call `s2_rebuild` on the geometry (think of this as a `st_make_valid` on the sphere)

Details

`st_as_s2` converts an sf POLYGON object into a form readable by s2.

Value

`sf_use_s2` returns the value of this variable before (re)setting it, invisibly if `use_s2` is not missing.

Examples

```r
m = rbind(c(-1,-1), c(1,-1), c(1,1), c(-1,1), c(-1,-1))
m1 = rbind(c(-1,-1), c(1,-1), c(1,1), c(-1,1), c(-1,0), c(-1,-1))
m0 = m[5:1,]
mp = st_multipolygon(list(
  list(m, 0.8 * m0, 0.01 * m1 + 0.9),
  list(0.7* m, 0.6*m0),
  list(0.5 * m0),
  list(m+2),
  list(m+4,(.9*m0)+4)
))
sf = st_sfc(mp, mp, crs = 'EPSG:4326')
s2 = st_as_s2(sf)
```

---

**sf**

*Create sf object*

Description

Create sf, which extends data.frame-like objects with a simple feature list column
Usage

```
st_sf(
  ..., 
  agr = NA_agr_, 
  row.names, 
  stringsAsFactors = sf_stringsAsFactors(), 
  crs, 
  precision, 
  sf_column_name = NULL, 
  check_ring_dir = FALSE, 
  sfc_last = TRUE
)
```

## S3 method for class 'sf'
```
x[i, j, ..., drop = FALSE, op = st_intersects]
```

## S3 method for class 'sf'
```
print(x, ..., n = getOption("sf_max_print", default = 10))
```

Arguments

```
...          column elements to be binded into an sf object or a single list or data.frame 
            with such columns; at least one of these columns shall be a geometry list-column 
            of class sfc or be a list-column that can be converted into an sfc by st_as_sfc.
agr          character vector; see details below.
row.names    row.names for the created sf object
stringsAsFactors
  logical; see st_read
crs          coordinate reference system, something suitable as input to st_crs
precision    numeric; see st_as_binary
sf_column_name
  character; name of the active list-column with simple feature geometries; in case
  there is more than one and sf_column_name is NULL, the first one is taken.
check_ring_dir see st_read
sfc_last      logical; if TRUE, sfc columns are always put last, otherwise column order is left
              unmodified.
x            object of class sf
i            record selection, see [.data.frame
j            variable selection, see [.data.frame
drop        logical, default FALSE; if TRUE drop the geometry column and return a data.frame, 
            else make the geometry sticky and return a sf object.
op           function; geometrical binary predicate function to apply when i is a simple 
            feature object
n            maximum number of features to print; can be set globally by options(sf_max_print=...)
```
Details

`agr`, attribute-geometry-relationship, specifies for each non-geometry attribute column how it relates to the geometry, and can have one of following values: "constant", "aggregate", "identity".

"constant" is used for attributes that are constant throughout the geometry (e.g. land use), "aggregate" where the attribute is an aggregate value over the geometry (e.g. population density or population count), "identity" when the attributes uniquely identifies the geometry of particular "thing", such as a building ID or a city name. The default value, `NA_agr_`, implies we don’t know.

When a single value is provided to `agr`, it is cascaded across all input columns; otherwise, a named vector like `c(feature1='constant', ...)` will set `agr` value to 'constant' for the input column named `feature1`. See `demo(nc)` for a worked example of this.

When confronted with a data.frame-like object, `st_sf` will try to find a geometry column of class `sfc`, and otherwise try to convert list-columns when available into a geometry column, using `st_as_sfc`.

`[.sf` will return a `data.frame` or vector if the geometry column (of class `sfc`) is dropped (`drop=TRUE`), an `sfc` object if only the geometry column is selected, and otherwise return an `sf` object; see also `[.data.frame` for `[.sf` arguments are passed to `op`.

Examples

```r
# SF  with points

g = st_sfc(st_point(1:2))
st_sf(a=3,g)
st_sf(g, a=3)
st_sf(a=3, st_sfc(st_point(1:2))) # better to name it!

# create empty structure with preallocated empty geometries:
nrows <- 10

geometry = st_sfc(lapply(1:nrows, function(x) st_geometrycollection()))
df <- st_sf(id = 1:nrows, geometry = geometry)
g = st_sfc(st_point(1:2), st_point(3:4))
s = st_sf(a=3:4, g)
s[1,]
class(s[1,])
s[1,]
class(s[1,])
s[2,]
class(s[2,])
g = st_sf(a=2:3, g)
pol = st_sfc(st_polygon(list(cbind(c(0,3,3,0,0),c(0,0,3,3,0)))))
h = st_sf(r = 5, pol)
g[h,]
h[g,]
```

<table>
<thead>
<tr>
<th>sf-defunct</th>
<th>Deprecated functions in sf</th>
</tr>
</thead>
</table>

Description

These functions are provided for compatibility with older version of `sf`. They may eventually be completely removed.
Usage

st_read_db(
  conn = NULL,
  table = NULL,
  query = NULL,
  geom_column = NULL,
  EWKB = TRUE,
  ...
)

Arguments

  conn          open database connection
  table         table name
  query         SQL query to select records; see details
  geom_column   deprecated. Geometry column name
  EWKB          logical; is the WKB of type EWKB? if missing, defaults to TRUE
  ...           parameter(s) passed on to st_as_sf

Details

The geom_column argument is deprecated. The function will automatically find the geometry type columns. For the RPostgreSQL drivers it will try to cast all the character columns, which can be long for very wide tables.

Details

  st_read_db   now a synonym for st_read
  st_write_db  now a synonym for st_write

sfc

Create simple feature geometry list column

Description

Create simple feature geometry list column, set class, and add coordinate reference system and precision

Usage

st_sfc(
...,
crs = NA_crs_,
precision = 0,
check_ring_dir = FALSE,
dim,
recompute_bbox = FALSE
)

Arguments

... zero or more simple feature geometries (objects of class sfg), or a single list of such objects; NULL values will get replaced by empty geometries.

crs coordinate reference system: integer with the EPSG code, or character with proj4string

precision numeric; see st_as_binary

check_ring_dir see st_read

dim character; if this function is called without valid geometries, this argument may carry the right dimension to set empty geometries

recompute_bbox logical; use TRUE to force recomputation of the bounding box

Details

A simple feature geometry list-column is a list of class c("stc_TYPE", "sfc") which most often contains objects of identical type; in case of a mix of types or an empty set, TYPE is set to the superclass GEOMETRY.

Value

an object of class sfc, which is a classed list-column with simple feature geometries.

Examples

pt1 = st_point(c(0,1))
pt2 = st_point(c(1,1))
(sfc = st_sfc(pt1, pt2))
d = st_sf(data.frame(a=1:2, geom=sfc))

---

**sf_extSoftVersion**

Provide the external dependencies versions of the libraries linked to sf

Description

Provide the external dependencies versions of the libraries linked to sf

Usage

sf_extSoftVersion()
**Description**

directly transform a set of coordinates

**Usage**

```r
sf_project()

sf_project(
  from = character(0),
  to = character(0),
  pts,
  keep = FALSE,
  warn = TRUE,
  authority_compliant = st_axis_order()
)
```

**Arguments**

- `from` character description of source CRS, or object of class `crs`, or pipeline describing a transformation
- `to` character description of target CRS, or object of class `crs`
- `pts` two-, three- or four-column numeric matrix, or object that can be coerced into a matrix; columns 3 and 4 contain z and t values.
- `keep` logical value controlling the handling of unprojectable points. If `keep` is `TRUE`, then such points will yield `Inf` or `-Inf` in the return value; otherwise an error is reported and nothing is returned.
- `warn` logical; if `TRUE`, warn when non-finite values are generated
- `authority_compliant` logical; `TRUE` means handle axis order authority compliant (e.g. EPSG:4326 implying x=lat, y=lon), `FALSE` means use visualisation order (i.e. always x=lon, y=lat)

**Details**

`sf_add_proj_units` loads the PROJ units ‘link’, ‘us_in’, ‘ind_yd’, ‘ind_ft’, and ‘ind_ch’ into the udnits database, and returns `TRUE` invisibly on success.

**Value**

two-column numeric matrix with transformed/converted coordinates, returning invalid values as `Inf`
Examples

sf_add_proj_units()

Methods for dealing with sparse geometry binary predicate lists

Description

Methods for dealing with sparse geometry binary predicate lists

Usage

## S3 method for class 'sgbp'
print(x, ..., n = 10, max_nb = 10)

## S3 method for class 'sgbp'
t(x)

## S3 method for class 'sgbp'
as.matrix(x, ...)

## S3 method for class 'sgbp'
dim(x)

Arguments

x       object of class sgbp
...
     ignored
n       integer; maximum number of items to print
max_nb  integer; maximum number of neighbours to print for each item

Details

sgbp are sparse matrices, stored as a list with integer vectors holding the ordered TRUE indices of each row. This means that for a dense, $m \times n$ matrix $Q$ and a list $L$, if $Q[i,j]$ is TRUE then $j$ is an element of $L[[i]]$. Reversed: when $k$ is the value of $L[[i]][j]$, then $Q[i,k]$ is TRUE.
Create simple feature from a numeric vector, matrix or list

Description

Create simple feature from a numeric vector, matrix or list

Usage

\[
\begin{align*}
st\_point(x & = c(NA\textunderscore real, NA\textunderscore real), \text{dim } = "XYZ") \\
st\_multipoint(x & = \text{matrix(numeric(0), 0, 2), dim } = "XYZ") \\
st\_linestring(x & = \text{matrix(numeric(0), 0, 2), dim } = "XYZ") \\
st\_polygon(x & = \text{list()}, \text{dim } = \text{if (length(x)) "XYZ" else "XY"}) \\
st\_multilinestring(x & = \text{list()}, \text{dim } = \text{if (length(x)) "XYZ" else "XY"}) \\
st\_multipolygon(x & = \text{list()}, \text{dim } = \text{if (length(x)) "XYZ" else "XY"}) \\
st\_geometrycollection(x & = \text{list()}, \text{dims } = "XY")
\end{align*}
\]

## S3 method for class 'sfg'
\[
\begin{align*}
\text{print}(x, \ldots, \text{width } = 0) \\
\text{head}(x, n = 10L, \ldots) \\
\text{format}(x, \ldots, \text{width } = 30) \\
\text{c}(\ldots, \text{recursive } = \text{FALSE, flatten } = \text{TRUE}) \\
\text{as.matrix}(x, \ldots)
\end{align*}
\]

Arguments

\begin{itemize}
\item \textbf{x} for \texttt{st\_point}, numeric vector (or one-row-matrix) of length 2, 3 or 4; for \texttt{st\_linestring} and \texttt{st\_multipoint}, numeric matrix with points in rows; for \texttt{st\_polygon} and \texttt{st\_multilinestring}, list with numeric matrices with points in rows; for \texttt{st\_multipolygon}, list of lists with numeric matrices; for \texttt{st\_geometrycollection} list with (non-geometrycollection) simple feature objects
\end{itemize}
dim character, indicating dimensions: "XY", "XYZ", "XYM", or "XYZM"; only really needed for three-dimensional points (which can be either XYZ or XYM) or empty geometries; see details
dims character; specify dimensionality in case of an empty (NULL) geometrycollection, in which case x is the empty list()
... objects to be pasted together into a single simple feature
width integer; number of characters to be printed (max 30; 0 means print everything)
n integer; number of elements to be selected
recursive logical; ignored
flatten logical; if TRUE, try to simplify results; if FALSE, return geometrycollection containing all objects

Details
"XYZ" refers to coordinates where the third dimension represents altitude, "XYM" refers to three-dimensional coordinates where the third dimension refers to something else ("M" for measure); checking of the sanity of x may be only partial.
When flatten=TRUE, this method may merge points into a multipoint structure, and may not preserve order, and hence cannot be reverted. When given fish, it returns fish soup.

Value
object of the same nature as x, but with appropriate class attribute set
as.matrix returns the set of points that form a geometry as a single matrix, where each point is a row; use unlist(x, recursive = FALSE) to get sets of matrices.

Examples

(p1 = st_point(c(1,2)))
class(p1)
st_bbox(p1)
(p2 = st_point(c(1,2,3)))
class(p2)
(p3 = st_point(c(1,2,3), "XYM"))
pts = matrix(1:10, , 2)
(mp1 = st_multipoint(pts))
pts = matrix(1:15, , 3)
(mp2 = st_multipoint(pts))
(mp3 = st_multipoint(pts, "XYM"))
pts = matrix(1:20, , 4)
(mp4 = st_multipoint(pts))
pts = matrix(1:10, , 2)
(ls1 = st_linestring(pts))
pts = matrix(1:15, , 3)
(ls2 = st_linestring(pts))
(ls3 = st_linestring(pts, "XYM"))
pts = matrix(1:20, , 4)
(ls4 = st_linestring(10, 2))
stars functions only exported to be used internally by stars
Description

functions only exported to be used internally by stars

Usage

.get_layout(bb, n, total_size, key.pos, key.length, mfrow = NULL)

.degAxis(side, at, labels, ..., lon, lat, ndiscr, reset)

.image_scale(
  z,
  col,
  breaks = NULL,
  key.pos,
  add.axis = TRUE,
  at = NULL,
  ...,  
  axes = FALSE,
  key.length,
  logz = FALSE,
  lab = ""
)

.image_scale_factor(
  z,
  col,
  key.pos,
  add.axis = TRUE,
  ...,  
  axes = FALSE,
  key.width,
  key.length
)

Arguments

bb              ignore
n               ignore
total_size      ignore
key.pos         ignore
key.length      ignore
mfrow           length-2 integer vector with number of rows, columns
side            ignore
at              ignore
labels          ignore
...             ignore
Description

get or set relation_to_geometry attribute of an sf object

Usage

NA Agr

st_agr(x, ...)

st_agr(x) <- value

st_set_agr(x, value)

Arguments

x object of class sf

... ignored

value character, or factor with appropriate levels; if named, names should correspond to the non-geometry list-column columns of x

Format

An object of class factor of length 1.

Details

NA Agr is the agr object with a missing value.
st_as_binary 

Convert sfc object to a WKB object

Description

Convert sfc object to an WKB object

Usage

st_as_binary(x, ...)

## S3 method for class 'sfc'
st_as_binary(
  x,
  ...,
  EWKB = FALSE,
  endian = .Platform$endian,
  pureR = FALSE,
  precision = attr(x, "precision"),
  hex = FALSE
)

## S3 method for class 'sfg'
st_as_binary(
  x,
  ...,
  endian = .Platform$endian,
  EWKB = FALSE,
  pureR = FALSE,
  hex = FALSE,
  srid = 0
)

Arguments

x 
object to convert

... 
ignored

EWKB 
logical; use EWKB (PostGIS), or (default) ISO-WKB?

endian 
character; either "big" or "little"; default: use that of platform

pureR 
logical; use pure R solution, or C++?

precision 
numeric; if zero, do not modify; to reduce precision: negative values convert to float (4-byte real); positive values convert to round(x*precision)/precision. See details.

hex 
logical; return as (unclassed) hexadecimal encoded character vector?

srid 
integer; override srid (can be used when the srid is unavailable locally).
st_as_grob

Details

st_as_binary is called on sfc objects on their way to the GDAL or GEOS libraries, and hence does rounding (if requested) on the fly before e.g. computing spatial predicates like st_intersects. The examples show a round-trip of an sfc to and from binary.

For the precision model used, see also https://locationtech.github.io/jts/javadoc/org/locationtech/jts/geom/PrecisionModel.html. There, it is written that: “... to specify 3 decimal places of precision, use a scale factor of 1000. To specify -3 decimal places of precision (i.e. rounding to the nearest 1000), use a scale factor of 0.001.”. Note that ALL coordinates, so also Z or M values (if present) are affected.

Examples

# examples of setting precision:
st_point(c(1/3, 1/6)) %>% st_sfc(precision = 1000) %>% st_as_binary %>% st_as_sfc
st_point(c(1/3, 1/6)) %>% st_sfc(precision = 100) %>% st_as_binary %>% st_as_sfc
st_point(1e6 * c(1/3, 1/6)) %>% st_sfc(precision = 0.01) %>% st_as_binary %>% st_as_sfc
st_point(1e6 * c(1/3, 1/6)) %>% st_sfc(precision = 0.001) %>% st_as_binary %>% st_as_sfc

st_as_grob

Convert sf* object to a grob

Description

Convert sf* object to an grid graphics object (grob)

Usage

st_as_grob(x, ...)

Arguments

x object to be converted into an object class grob

... passed on to the xxxGrob function, e.g. gp = gpar(col = 'red')

st_as_sf

Convert foreign object to an sf object

Description

Convert foreign object to an sf object
Usage

```r
st_as_sf(x, ...)
```

## S3 method for class 'data.frame'
```r
st_as_sf(
  x,
  ...,
  agr = NA_agr_,
  coords,
  wkt,
  dim = "XYZ",
  remove = TRUE,
  na.fail = TRUE,
  sf_column_name = NULL
)
```

## S3 method for class 'sf'
```r
st_as_sf(x, ...)
```

## S3 method for class 'sfc'
```r
st_as_sf(x, ...)
```

## S3 method for class 'Spatial'
```r
st_as_sf(x, ...)
```

## S3 method for class 'map'
```r
st_as_sf(x, ..., fill = TRUE, group = TRUE)
```

## S3 method for class 'ppp'
```r
st_as_sf(x, ...)
```

## S3 method for class 'psp'
```r
st_as_sf(x, ...)
```

## S3 method for class 'lpp'
```r
st_as_sf(x, ...)
```

## S3 method for class 's2_geography'
```r
st_as_sf(x, ..., crs = st_crs(4326))
```

Arguments

- **x** object to be converted into an object class sf
- **...** passed on to `st_sf`, might included named arguments `crs` or `precision`
- **agr** character vector; see details section of `st_sf`
- **coords** in case of point data: names or numbers of the numeric columns holding coordinates
**st_as_sf**

- **wkt**: name or number of the character column that holds WKT encoded geometries
- **dim**: passed on to `st_point` (only when argument `coords` is given)
- **remove**: logical; when `coords` or `wkt` is given, remove these columns from data.frame?
- **na.fail**: logical; if TRUE, raise an error if coordinates contain missing values
- **sf_column_name**: character; name of the active list-column with simple feature geometries; in case there is more than one and `sf_column_name` is NULL, the first one is taken.
- **fill**: logical; the value for fill that was used in the call to `map`.
- **group**: logical; if TRUE, group id labels from `map` by their prefix before :.
- **crs**: coordinate reference system to be assigned; object of class `crs`

**Details**

setting argument `wkt` annihilates the use of argument `coords`. If `x` contains a column called "geometry", `coords` will result in overwriting of this column by the `sfc` geometry list-column. Setting `wkt` will replace this column with the geometry list-column, unless `remove` is FALSE.

**Examples**

```r
pt1 = st_point(c(0,1))
pt2 = st_point(c(1,1))
st_sfc(pt1, pt2)
d = data.frame(a = 1:2)
d$geom = st_sfc(pt1, pt2)
if (require(sp, quietly = TRUE)) {
  df = st_as_sf(d)
d$geom = c("POINT(0 0)", "POINT(0 1)")
df = st_as_sf(d, wkt = "geom")
d$geom2 = st_sfc(pt1, pt2)
st_as_sf(d) # should warn
}
if (require(sp, quietly = TRUE)) {
  data(meuse, package = "sp")
  meuse_sf = st_as_sf(meuse, coords = c("x", "y"), crs = 28992, agr = "constant")
  meuse_sf[1:3,]
  summary(meuse_sf)
}
x = rbind(c(-1,-1), c(1,-1), c(1,1), c(-1,1), c(-1,-1))
x1 = 0.1 * x + 0.1
x2 = 0.1 * x + 0.4
x3 = 0.1 * x + 0.7
y = x + 3
y1 = x1 + 3
y3 = x3 + 3
m = matrix(c(3, 0), 5, 2, byrow = TRUE)
z = x + m
z1 = x1 + m
z2 = x2 + m
z3 = x3 + m
p1 = Polygons(list( Polygon(x[5:1,]), Polygon(x2), Polygon(x3),
Polygons(list( Polygon(y[5:1,]), Polygon(y1), Polygon(x1), Polygon(y3)), "ID1")
```

p2 = Polygons(list( Polygon(z[5:1,]), Polygon(z2), Polygon(z3), Polygon(z1)), 
"ID2")
r = SpatialPolygons(list(p1,p2))
a = suppressWarnings(st_as_sf(r))
summary(a)
demo(meuse, ask = FALSE, echo = FALSE)
summary(st_as_sf(meuse))
summary(st_as_sf(meuse.grid))
summary(st_as_sf(meuse.area))
summary(st_as_sf(meuse.riv))
summary(st_as_sf(as(meuse.riv, "SpatialLines")))
pol.grd = as(meuse.grid, "SpatialPolygonsDataFrame")
# summary(st_as_sf(pol.grd))
# summary(st_as_sf(as(pol.grd, "SpatialLinesDataFrame")))
if (require(spatstat.geom)) {
  g = st_as_sf(gorillas)
  # select only the points:
  g[st_is(g, "POINT"),]
}
if (require(spatstat.linnet)) {
  data(chicago)
  plot(st_as_sf(chicago)["label"])
  plot(st_as_sf(chicago)[-1,"label"])
}

---

**st_as_sfc**

*Convert foreign geometry object to an sfc object*

**Description**

Convert foreign geometry object to an sfc object

**Usage**

```r
## S3 method for class 'pq_geometry'
st_as_sfc(
  x,
  ...,
  EWKB = TRUE,
  spatialite = FALSE,
  pureR = FALSE,
  crs = NA_crs_
)

## S3 method for class 'list'
st_as_sfc(x, ...,

## S3 method for class 'blob'
```
st_as_sfc(x, ...)

## S3 method for class 'bbox'
st_as_sfc(x, ...)

## S3 method for class 'WKB'
st_as_sfc(
  x,
  ...,
  EWKB = FALSE,
  spatialite = FALSE,
  pureR = FALSE,
  crs = NA_crs_
)

## S3 method for class 'raw'
st_as_sfc(x, ...)

## S3 method for class 'character'
st_as_sfc(x, crs = NA_integer_, ..., GeoJSON = FALSE)

## S3 method for class 'factor'
st_as_sfc(x, ...)

## S3 method for class 'SpatialPoints'
st_as_sfc(x, ..., precision = 0)

## S3 method for class 'SpatialPixels'
st_as_sfc(x, ..., precision = 0)

## S3 method for class 'SpatialMultiPoints'
st_as_sfc(x, ..., precision = 0)

## S3 method for class 'SpatialLines'
st_as_sfc(x, ..., precision = 0, forceMulti = FALSE)

## S3 method for class 'SpatialPolygons'
st_as_sfc(x, ..., precision = 0, forceMulti = FALSE)

## S3 method for class 'map'
st_as_sfc(x, ...)

## S3 method for class 's2_geography'
st_as_sfc(
  x,
  ...,
crs = st_crs(4326),
endian = match(.Platform$endian, c("big", "little")) - 1L
)

Arguments

- **x**
  - object to convert

- ... further arguments

- **EWKB**
  - logical; if TRUE, parse as EWKB (extended WKB; PostGIS: ST_AsEWKB),
    otherwise as ISO WKB (PostGIS: ST_AsBinary)

- **spatialite**
  - logical; if TRUE, WKB is assumed to be in the spatialite dialect, see https://www.gaia-gis.it/gaia-sins/BLOB-Geometry.html; this is only supported in native endian-ness (i.e., files written on system with the same endian-ness as that on which it is being read).

- **pureR**
  - logical; if TRUE, use only R code, if FALSE, use compiled (C++) code; use TRUE when the endian-ness of the binary differs from the host machine (.Platform$endian).

- **crs**
  - coordinate reference system to be assigned; object of class `crs`

- **GeoJSON**
  - logical; if TRUE, try to read geometries from GeoJSON text strings geometry, see `st_crs()`

- **precision**
  - precision value; see `st_as_binary`

- **forceMulti**
  - logical; if TRUE, force coercion into `MULTIPOLYGON` or `MULTILINE` objects, else autodetect

- **endian**
  - integer; 0 or 1: defaults to the endian of the native machine

Details

When converting from WKB, the object x is either a character vector such as typically obtained from PostGIS (either with leading "0x" or without), or a list with raw vectors representing the features in binary (raw) form.

If x is a character vector, it should be a vector containing well-known-text, or Postgis EWKT or GeoJSON representations of a single geometry for each vector element.

If x is a factor, it is converted to character.

Examples

```r
wkb = structure(list("01010000204071000000000000801A06410000000AC5C1441"), class = "WKB")
st_as_sfc(wkb, EWKB = TRUE)

wkb = structure(list("0x0101000020407100000000000000801A06410000000AC5C1441"), class = "WKB")
st_as_sfc(wkb, EWKB = TRUE)

st_as_sfc(st_as_binary(st_sfc(st_point(0:1)))[[1]], crs = 4326)
st_as_sfc("SRID=3978;LINESTRING(1663106 -105415,1664320 -104617)"")
```
st_as_text

Return Well-known Text representation of simple feature geometry or coordinate reference system

Description

Return Well-known Text representation of simple feature geometry or coordinate reference system

Usage

```r
## S3 method for class 'crs'
st_as_text(x, ..., projjson = FALSE, pretty = FALSE)

st_as_text(x, ...)

## S3 method for class 'sfg'
st_as_text(x, ...)

## S3 method for class 'sfc'
st_as_text(x, ..., EWKT = FALSE)
```

Arguments

- `x`: object of class `sfg`, `sfc` or `crs`
- `...`: modifiers; in particular digits can be passed to control the number of digits used
- `projjson`: logical; if TRUE, return projjson form (requires GDAL 3.1 and PROJ 6.2), else return well-known-text form
- `pretty`: logical; if TRUE, print human-readable well-known-text representation of a coordinate reference system
- `EWKT`: logical; if TRUE, print SRID=xxx; before the WKT string if `epsg` is available

Details

The returned WKT representation of simple feature geometry conforms to the simple features access specification and extensions (known as EWKT, supported by PostGIS and other simple features implementations for addition of a SRID to a WKT string).

Examples

```r
st_as_text(st_point(1:2))
st_as_text(st_sfc(st_point(c(-90,40)), crs = 4326), EWKT = TRUE)
```
Return bounding of a simple feature or simple feature set

Description
Return bounding of a simple feature or simple feature set

Usage

```r
## S3 method for class 'bbox'
is.na(x)
st_bbox(obj, ...)

## S3 method for class 'POINT'
st_bbox(obj, ...)

## S3 method for class 'MULTIPOINT'
st_bbox(obj, ...)

## S3 method for class 'LINESTRING'
st_bbox(obj, ...)

## S3 method for class 'POLYGON'
st_bbox(obj, ...)

## S3 method for class 'MULTILINESTRING'
st_bbox(obj, ...)

## S3 method for class 'MULTIPOLYGON'
st_bbox(obj, ...)

## S3 method for class 'GEOMETRYCOLLECTION'
st_bbox(obj, ...)

## S3 method for class 'MULTISURFACE'
st_bbox(obj, ...)

## S3 method for class 'MULTICURVE'
st_bbox(obj, ...)

## S3 method for class 'CURVEPOLYGON'
st_bbox(obj, ...)

## S3 method for class 'COMPOUNDCURVE'
st_bbox(obj, ...)
```
## S3 method for class 'POLYHEDRALSURFACE'

```
st_bbox(obj, ...)
```

## S3 method for class 'TIN'

```
st_bbox(obj, ...)
```

## S3 method for class 'TRIANGLE'

```
st_bbox(obj, ...)
```

## S3 method for class 'CIRCULARSTRING'

```
st_bbox(obj, ...)
```

## S3 method for class 'sfc'

```
st_bbox(obj, ...)
```

## S3 method for class 'sf'

```
st_bbox(obj, ...)
```

## S3 method for class 'Spatial'

```
st_bbox(obj, ...)
```

## S3 method for class 'Raster'

```
st_bbox(obj, ...)
```

## S3 method for class 'Extent'

```
st_bbox(obj, ..., crs = NA_crs_)
```

## S3 method for class 'numeric'

```
st_bbox(obj, ..., crs = NA_crs_)
```

NA_bbox_

## S3 method for class 'bbox'

```
format(x, ...)
```

### Arguments

- **x**
  - object of class bbox

- **obj**
  - object to compute the bounding box from

- **...**
  - for format.bbox, passed on to format to format individual numbers

- **crs**
  - object of class crs, or argument to st_crs, specifying the CRS of this bounding box.

### Format

An object of class bbox of length 4.
Details

NA_bbox_ represents the missing value for a bbox object

Value

a numeric vector of length four, with xmin, ymin, xmax and ymax values; if obj is of class sf, sfc, Spatial or Raster, the object returned has a class bbox, an attribute crs and a method to print the bbox and an st_crs method to retrieve the coordinate reference system corresponding to obj (and hence the bounding box). st_as_sfc has a methods for bbox objects to generate a polygon around the four bounding box points.

Examples

```r
a = st_sf(a = 1:2, geom = st_sfc(st_point(0:1), st_point(1:2)), crs = 4326)
st_bbox(a)
st_as_sfc(st_bbox(a))
st_bbox(c(xmin = 16.1, xmax = 16.6, ymax = 48.6, ymin = 47.9), crs = st_crs(4326))
```

---

Description

Longitudes can be broken at the antimeridian of a target central longitude to permit plotting of (usually world) line or polygon objects centred on the chosen central longitude. The method may only be used with non-projected, geographical coordinates and linestring or polygon objects. s2 is turned off internally to permit the use of a rectangular bounding box. If the input geometries go outside [-180, 180] degrees longitude, the protruding geometries will also be split using the same tol= values; in this case empty geometries will be dropped first.

Usage

```r
st_break_antimeridian(x, lon_0 = 0, tol = 1e-04, ...)
```

## S3 method for class 'sf'
```r
st_break_antimeridian(x, lon_0 = 0, tol = 1e-04, ...)
```

## S3 method for class 'sfc'
```r
st_break_antimeridian(x, lon_0 = 0, tol = 1e-04, ...)
```

Arguments

- **x**: object of class sf or sfc
- **lon_0**: target central longitude (degrees)
- **tol**: half of break width (degrees, default 0.0001)
- **...**: ignored here
### Examples

```r
if (require("maps", quietly=TRUE)) {
  opar = par(mfrow=c(3, 2))
  wld = st_as_sf(map(fill=FALSE, interior=FALSE, plot=FALSE), fill=FALSE)
  for (lon_0 in c(-170, -90, -10, 10, 90, 170)) {
    wld |> st_break_antimeridian(lon_0=lon_0) |>
      st_transform(paste0("+proj=natearth +lon_0=", lon_0)) |>
      st_geometry() |> plot(main=lon_0)
  }
  par(opar)
}
```

---

**st_cast**  
*Cast geometry to another type: either simplify, or cast explicitly*

### Description

Cast geometry to another type: either simplify, or cast explicitly

### Usage

```r
## S3 method for class 'MULTIPOLYGON'
st_cast(x, to, ...)

## S3 method for class 'MULTILINESTRING'
st_cast(x, to, ...)

## S3 method for class 'MULTIPOINT'
st_cast(x, to, ...)

## S3 method for class 'POLYGON'
st_cast(x, to, ...)

## S3 method for class 'LINESTRING'
st_cast(x, to, ...)

## S3 method for class 'POINT'
st_cast(x, to, ...)

## S3 method for class 'GEOMETRYCOLLECTION'
st_cast(x, to, ...)

## S3 method for class 'CIRCULARSTRING'
st_cast(x, to, ...)
```
## S3 method for class 'MULTISURFACE'
st_cast(x, to, ...)

## S3 method for class 'COMPOUNDCURVE'
st_cast(x, to, ...)

## S3 method for class 'MULTICURVE'
st_cast(x, to, ...)

## S3 method for class 'CURVE'
st_cast(x, to, ...)

st_cast(x, to, ...)

## S3 method for class 'sfc'
st_cast(x, to, ..., ids = seq_along(x), group_or_split = TRUE)

## S3 method for class 'sf'
st_cast(x, to, ..., warn = TRUE, do_split = TRUE)

## S3 method for class 'sfc_CIRCULARSTRING'
st_cast(x, to, ...)

### Arguments

- **x**: object of class sfg, sfc or sf
- **to**: character; target type, if missing, simplification is tried; when x is of type sfg (i.e., a single geometry) then to needs to be specified.
- **...**: ignored
- **ids**: integer vector, denoting how geometries should be grouped (default: no grouping)
- **group_or_split**: logical; if TRUE, group or split geometries; if FALSE, carry out a 1-1 per-geometry conversion.
- **warn**: logical; if TRUE, warn if attributes are assigned to sub-geometries
- **do_split**: logical; if TRUE, allow splitting of geometries in sub-geometries

### Details

When converting a GEOMETRYCOLLECTION to COMPOUNDCURVE, MULTISURFACE or CURVEPOLYGON, the user is responsible for the validity of the resulting object: no checks are being carried out by the software.

When converting mixed, GEOMETRY sets, it may help to first convert to the MULTI-type, see examples

the st_cast method for sf objects can only split geometries, e.g. cast MULTIPOLYGON into multiple POLYGON features. In case of splitting, attributes are repeated and a warning is issued when non-constant attributes are assigned to sub-geometries. To merge feature geometries and attribute values, use aggregate or summarise.
**Value**

object of class to if successful, or unmodified object if unsuccessful. If information gets lost while type casting, a warning is raised.

In case to is missing, `st_cast.sfc` will coerce combinations of "POINT" and "MULTIPOINT", "LINESTRING" and "MULTILINESTRING", "POLYGON" and "MULTIPOLYGON" into their "MULTI..." form, or in case all geometries are "GEOMETRYCOLLECTION" will return a list of all the contents of the "GEOMETRYCOLLECTION" objects, or else do nothing. In case to is specified, if to is "GEOMETRY", geometries are not converted, else, `st_cast` will try to coerce all elements into to; ids may be specified to group e.g. "POINT" objects into a "MULTIPOINT", if not specified no grouping takes place. If e.g. a "sfc_MULTIPOINT" is cast to a "sfc_POINT", the objects are split, so no information gets lost, unless `group_or_split` is FALSE.

**Examples**

```r
# example(st_read)
c = read(system.file("shape/nc.shp", package="sf"))
mpl <- c$geometry[[4]]
#st_cast(x) ## error 'argument "to" is missing, with no default'
cast_all <- function(xg) {
  lapply(c("MULTIPOLYGON", "MULTILINESTRING", "MULTIPOINT", "POLYGON", "LINESTRING", "POINT"),
    function(x) {st_cast(xg, x))
  }
}
st_sfc(cast_all(mpl))
```

```r
p1 <- structure(c(0, 1, 3, 2, 1, 0, 0, 2, 4, 4, 0), .Dim = c(6L, 2L))
p2 <- structure(c(1, 1, 2, 1, 1, 2, 2, 1), .Dim = c(4L, 2L))
st_polygon(list(p1, p2))
mls <- st_cast(nc$geometry[[4]], "MULTILINESTRING")
st_sfc(cast_all(mls))
mpt <- st_cast(nc$geometry[[4]], "MULTIPOINT")
st_sfc(cast_all(mpt))
pl <- st_cast(nc$geometry[[4]], "POLYGON")
st_sfc(cast_all(pl))
ls <- st_cast(nc$geometry[[4]], "LINESTRING")
st_sfc(cast_all(ls))
pt <- st_cast(nc$geometry[[4]], "POINT")
## st_sfc(cast_all(pt)) ## Error: cannot create MULTIPOLYGON from POINT
s = st_multipoint(rbind(c("POINT", "MULTIPOINT"), function(x) st_cast(pt, x))
```

```r
https://github.com/r-spatial/sf/issues/1930:
```

```r
pt1 <- st_point(c(0, 1))
pt23 <- st_multipoint(matrix(c(1,2,3,4), ncol = 2, byrow = TRUE))
d <- st_sf(geom = st_sfc(pt1, pt23))
st_cast(d, "POINT") # will not convert the entire MULTIPOINT, and warns
```
st_cast(d, "MULTIPOINT") %>% st_cast("POINT")

---

**st_cast_sfc_default**  
*Coerce geometry to MULTI* geometry

**Description**
Mixes of POINTS and MULTIPOINTS, LINESTRING and MULTILINESTRING, POLYGON and MULTIPOLYGON are returned as MULTIPOINTS, MULTILINESTRING and MULTIPOLYGONS respectively

**Usage**

```r
st_cast_sfc_default(x)
```

**Arguments**

- `x` list of geometries or simple features

**Details**
Geometries that are already MULTI* are left unchanged. Features that can’t be cast to a single MULTI* geometry are return as a GEOMETRYCOLLECTION

---

**st_collection_extract**  
*Given an object with geometries of type GEOMETRY or GEOMETRYCOLLECTION, return an object consisting only of elements of the specified type.*

**Description**
Similar to ST_CollectionExtract in PostGIS. If there are no sub-geometries of the specified type, an empty geometry is returned.

**Usage**

```r
st_collection_extract(
  x,
  type = c("POLYGON", "POINT", "LINESTRING"),
  warn = FALSE
)
```

```r
## S3 method for class 'sfg'
st_collection_extract(
  x,
  type = c("POLYGON", "POINT", "LINESTRING"),
)"
## Arguments

- **x**
  
  an object of class sf, sfc or sfg that has mixed geometry (GEOMETRY or GEOMETRYCOLLECTION).

- **type**
  
  character; one of "POLYGON", "POINT", "LINESTRING"

- **warn**
  
  logical; if TRUE, warn if attributes are assigned to sub-geometries when casting (see `st_cast`)

## Value

An object having the same class as `x`, with geometries consisting only of elements of the specified type. For sfg objects, an sfg object is returned if there is only one geometry of the specified type, otherwise the geometries are combined into an sfc object of the relevant type. If any subgeometries in the input are MULTI, then all of the subgeometries in the output will be MULTI.

## Examples

```
pt <- st_point(c(1, 0))
ls <- st_linestring(matrix(c(4, 3, 0, 0), ncol = 2))
poly1 <- st_polygon(list(matrix(c(5.5, 7, 7, 6, 5.5, 0, 0, -0.5, -0.5, 0), ncol = 2)))
poly2 <- st_polygon(list(matrix(c(6.6, 8, 8, 7, 6.6, 1, 1, 1.5, 1.5, 1), ncol = 2)))
multipoly <- st_multipolygon(list(poly1, poly2))

i <- st_geometrycollection(list(pt, ls, poly1, poly2))
j <- st_geometrycollection(list(pt, ls, poly1, poly2, multipoly))

st_collection_extract(i, "POLYGON")
st_collection_extract(i, "POINT")
st_collection_extract(i, "LINESTRING")

## A GEOMETRYCOLLECTION

aa <- rbind(st_sf(a=1, geom = st_sfc(i)),
            st_sf(a=2, geom = st_sfc(j)))
```
## With sf objects
st_collection_extract(aa, "POLYGON")
st_collection_extract(aa, "LINESTRING")
st_collection_extract(aa, "POINT")

## With sfc objects
st_collection_extract(st_geometry(aa), "POLYGON")
st_collection_extract(st_geometry(aa), "LINESTRING")
st_collection_extract(st_geometry(aa), "POINT")

## A GEOMETRY of single types
bb <- rbind(
st_sf(a = 1, geom = st_sfc(pt)),
st_sf(a = 2, geom = st_sfc(ls)),
st_sf(a = 3, geom = st_sfc(poly1)),
st_sf(a = 4, geom = st_sfc(multipoly))
)
st_collection_extract(bb, "POLYGON")

## A GEOMETRY of mixed single types and GEOMETRYCOLLECTIONS
cc <- rbind(aa, bb)
st_collection_extract(cc, "POLYGON")

---

### st_coordinates

retire coordinates in matrix form

#### Description

retrieve coordinates in matrix form

#### Usage

st_coordinates(x, ...)

#### Arguments

- **x**
  - object of class sf, sfc or sfg
- **...**
  - ignored

#### Value

matrix with coordinates (X, Y, possibly Z and/or M) in rows, possibly followed by integer indicators L1,...,L3 that point out to which structure the coordinate belongs; for POINT this is absent (each coordinate is a feature), for LINESTRING L1 refers to the feature, for MULTILINESTRING L1 refers to the part and L2 to the simple feature, for POLYGON L1 refers to the main ring or holes and L2 to
the simple feature, for MULTIPOLYGON L1 refers to the main ring or holes, L2 to the ring id in the MULTIPOLYGON, and L3 to the simple feature.

For POLYGONS, L1 can be used to identify exterior rings and inner holes. The exterior ring is when L1 is equal to 1. Interior rings are identified when L1 is greater than 1. L2 can be used to differentiate between the feature. Whereas for MULTIPOLYGON, L3 refers to the MULTIPOLYGON feature and L2 refers to the component POLYGON.

---

**st_crop**

crop an sf object to a specific rectangle

**Description**

crop an sf object to a specific rectangle

**Usage**

```r
st_crop(x, y, ...)  
```

```r  
# S3 method for class 'sfc'  
st_crop(x, y, ..., xmin, ymin, xmax, ymax)  
```

```r  
# S3 method for class 'sf'  
st_crop(x, y, ...)  
```

**Arguments**

- `x` object of class sf or sfc
- `y` numeric vector with named elements xmin, ymin, xmax and ymax, or object of class bbox, or object for which there is an `st_bbox` method to convert it to a bbox object
- `...` ignored
- `xmin` minimum x extent of cropping area
- `ymin` minimum y extent of cropping area
- `xmax` maximum x extent of cropping area
- `ymax` maximum y extent of cropping area

**Details**

setting arguments xmin, ymin, xmax and ymax implies that argument y gets ignored.
Examples

```r
box = c(xmin = 0, ymin = 0, xmax = 1, ymax = 1)
pol = st_sfc(st_buffer(st_point(c(.5, .5)), .6))
pol_sf = st_sf(a=1, geom=pol)
plot(st_crop(pol, box))
plot(st_crop(pol_sf, st_bbox(box)))
# alternative:
plot(st_crop(pol, xmin = 0, ymin = 0, xmax = 1, ymax = 1))
```

---

**st_crs**

Retrieve coordinate reference system from object

**Description**

Retrieve coordinate reference system from sf or sfc object

Set or replace retrieve coordinate reference system from object

**Usage**

```r
st_crs(x, ...)
```

## S3 method for class 'sf'
```r
st_crs(x, ...)
```

## S3 method for class 'numeric'
```r
st_crs(x, ...)
```

## S3 method for class 'character'
```r
st_crs(x, ...)
```

## S3 method for class 'sfc'
```r
st_crs(x, ..., parameters = FALSE)
```

## S3 method for class 'bbox'
```r
st_crs(x, ...)
```

## S3 method for class 'CRS'
```r
st_crs(x, ...)
```

## S3 method for class 'crs'
```r
st_crs(x, ...)
```

```r
st_crs(x) <- value
```

## S3 replacement method for class 'sf'
```r
st_crs(x) <- value
```
st_crs

## S3 replacement method for class 'sfc'

```r
st_crs(x) <- value
```

```r
st_set_crs(x, value)
```

NA_crs_

## S3 method for class 'crs'

```r
is.na(x)
```

## S3 method for class 'crs'

```r
x$name
```

## S3 method for class 'crs'

```r
format(x, ...)
```

```r
st_axis_order(authority_compliant = logical(0))
```

### Arguments

- **x**: numeric, character, or object of class `sf` or `sfc`
- **...**: ignored
- **parameters**: logical; FALSE by default; if TRUE return a list of coordinate reference system parameters, with named elements `SemiMajor`, `InvFlattening`, `units_gdal`, `IsVertical`, `WktPretty`, and `Wkt`
- **value**: one of (i) character: a string accepted by GDAL, (ii) integer, a valid EPSG value (numeric), or (iii) an object of class `crs`.
- **name**: element name
- **authority_compliant**: logical; specify whether axis order should be handled compliant to the authority; if omitted, the current value is printed.

### Format

An object of class `crs` of length 2.

### Details

The *crs functions create, get, set or replace the crs attribute of a simple feature geometry list-column. This attribute is of class `crs`, and is a list consisting of `input` (user input, e.g. "EPSG:4326" or "WGS84" or a proj4string), and `wkt`, an automatically generated wkt2 representation of the crs. If `x` is identical to the wkt2 representation, and the CRS has a name, this name is used for the `input` field.

Comparison of two objects of class `crs` uses the GDAL function `OGRSpatialReference::IsSame`. In case a coordinate reference system is replaced, no transformation takes place and a warning is raised to stress this.

NA_crs_ is the `crs` object with missing values for `input` and `wkt`. 
the $ method for crs objects retrieves named elements using the GDAL interface; named elements include "SemiMajor", "SemiMinor", "InvFlattening", "IsGeographic", "units_gdal", "IsVertical", "WktPretty", "Wkt", "Name", "proj4string", "epsg", "yx", "ud_unit", and axes (this may be subject to changes in future GDAL versions). "ud_unit" returns a valid units object or NULL if units are missing.

format.crs returns NA if the crs is missing valued, or else the name of a crs if it is different from "unknown", or else the user input if it was set, or else its "proj4string" representation;

st_axis_order can be used to get and set the axis order: TRUE indicates axes order according to the authority (e.g. EPSG:4326 defining coordinates to be latitude,longitude pairs), FALSE indicates the usual GIS (display) order (longitude,latitude). This can be useful when data are read, or have to be written, with coordinates in authority compliant order. The return value is the current state of this (FALSE, by default).

Value

If x is numeric, return crs object for EPSG:x; if x is character, return crs object for x; if x is of class sf or sfc, return its crs object.

Object of class crs, which is a list with elements input (length-1 character) and wkt (length-1 character). Elements may be NA valued; if all elements are NA the CRS is missing valued, and coordinates are assumed to relate to an arbitrary Cartesian coordinate system.

st_axis_order returns the (logical) current value if called without argument, or (invisibly) the previous value if it is being set.

Examples

```r
sfc = st_sfc(st_point(c(0,0)), st_point(c(1,1)))
sf = st_sf(a = 1:2, geom = sfc)
st_crs(sf) = 4326
st_geometry(sf)
sfc = st_sfc(st_point(c(0,0)), st_point(c(1,1)))
st_crs(sfc) = 4326
sfc
sfc = st_sfc(st_point(c(0,0)), st_point(c(1,1)))
sfc %>% st_set_crs(4326) %>% st_transform(3857)
st_crs("EPSG:3857")$input
st_crs(3857)$proj4string
st_crs(3857)$b # numeric
st_crs(3857)$units # character
pt = st_sfc(st_point(c(0, 60)), crs = 4326)
# st_axis_order() only has effect in GDAL >= 2.5.0:
st_axis_order() # query default: FALSE means interpret pt as (longitude latitude)
st_transform(pt, 3857)[[1]]
old_value = FALSE
if (sf_extSoftVersion()["GDAL"] >= "2.5.0")
  (old_value = st_axis_order(TRUE))
# now interpret pt as (latitude longitude), as EPSG:4326 prescribes:
st_axis_order() # query current value
st_transform(pt, 3857)[[1]]
st_axis_order(old_value) # set back to old value
```
Description

Get a list of the available GDAL drivers

Usage

st_drivers(what = "vector", regex)

Arguments

what character: "vector" or "raster", anything else will return all drivers.
regex character; regular expression to filter the name and long_name fields on

Details

The drivers available will depend on the installation of GDAL/OGR, and can vary; the st_drivers() function shows all the drivers that are readable, and which may be written. The field vsi refers to the driver’s capability to read/create datasets through the VSI*L API. See GDAL website for additional details on driver support.

Value

A data.frame with driver metadata.

Examples

# The following driver lists depend on the GDAL setup and platform used:
st_drivers()
st_drivers("raster", "GeoT")

st_geometry

Get, set, replace or rename geometry from an sf object

Description

Get, set, replace or rename geometry from an sf object
Usage

## S3 method for class 'sfc'
st_geometry(obj, ...)

st_geometry(obj, ...)

## S3 method for class 'sf'
st_geometry(obj, ...)

## S3 method for class 'sfc'
st_geometry(obj, ...)

## S3 method for class 'sfg'
st_geometry(obj, ...)

st_geometry(x) <- value

st_set_geometry(x, value)

st_drop_geometry(x, ...)

## S3 method for class 'sf'
st_drop_geometry(x, ...)

## Default S3 method:
st_drop_geometry(x, ...)

Arguments

- **obj**: object of class sf or sfc
- **...**: ignored
- **x**: object of class data.frame or sf
- **value**: object of class sfc, or character to set, replace, or rename the geometry of x

Details

when applied to a data.frame and when value is an object of class sfc, st_set_geometry and st_geometry<- will first check for the existence of an attribute sf_column and overwrite that, or else look for list-columns of class sfc and overwrite the first of that, or else write the geometry list-column to a column named geometry. In case value is character and x is of class sf, the “active” geometry column is set to x[[value]].

the replacement function applied to sf objects will overwrite the geometry list-column, if value is NULL, it will remove it and coerce x to a data.frame.

if x is of class sf, st_drop_geometry drops the geometry of its argument, and reclasses it accordingly; otherwise it returns x unmodified.
**Example**

```r
df = data.frame(a = 1:2)
sfc = st_sfc(st_point(c(3,4)), st_point(c(10,11)))
st_geometry(sfc)
st_geometry(df) <- sfc
class(df)
st_geometry(df)
st_geometry(df) <- sfc # replaces
st_geometry(df) <- NULL # remove geometry, coerce to data.frame
sf <- st_set_geometry(df, sfc) # set geometry, return sf
st_set_geometry(sf, NULL) # remove geometry, coerce to data.frame
```
st_graticule

Compute graticules and their parameters

Description

Compute graticules and their parameters

Usage

st_graticule(
  x = c(-180, -90, 180, 90),
  crs = st_crs(x),
  datum = st_crs(4326),
  ..., 
  lon = NULL,
  lat = NULL,
  ndiscr = 100,
  margin = 0.001
)

Arguments

x object of class sf, sfc or sfg or numeric vector with bounding box given as
(minx, miny, maxx, maxy).

 CRS object of class crs, with the display coordinate reference system

datum either an object of class crs with the coordinate reference system for the graticules, or NULL in which case a grid in the coordinate system of x is drawn, or NA, in which case an empty sf object is returned.

... ignored

lon numeric; values in degrees East for the meridians, associated with datum

lat numeric; values in degrees North for the parallels, associated with datum

ndiscr integer; number of points to discretize a parallel or meridian

margin numeric; small number to trim a longlat bounding box that touches or crosses +/-180 long or +/-90 latitude.

Value

an object of class sf with additional attributes describing the type (E: meridian, N: parallel) degree value, label, start and end coordinates and angle; see example.

Use of graticules

In cartographic visualization, the use of graticules is not advised, unless the graphical output will be used for measurement or navigation, or the direction of North is important for the interpretation of
the content, or the content is intended to display distortions and artifacts created by projection. Unnecessary use of graticules only adds visual clutter but little relevant information. Use of coastlines, administrative boundaries or place names permits most viewers of the output to orient themselves better than a graticule.

Examples

```r
library(sf)
if (require(maps, quietly = TRUE)) {
  usa = st_as_sf(map('usa', plot = FALSE, fill = TRUE))
  laea = st_crs("+proj=laea +lat_0=30 +lon_0=-95") # Lambert equal area
  usa <- st_transform(usa, laea)

  bb = st_bbox(usa)
  bbox = st_linestring(rbind(c(bb[1],bb[2]),c(bb[3],bb[2]),
                             c(bb[3],bb[4]),c(bb[1],bb[4]),c(bb[1],bb[2])))

  g = st_graticule(usa)
  plot(usa, xlim = 1.2 * c(-2450853.4, 2186391.9))
  plot(g$ll[1], add = TRUE, col = 'grey')
  plot(bbox, add = TRUE)
  points(g$x_start, g$y_start, col = 'red')
  points(g$x_end, g$y_end, col = 'blue')

  invisible(lapply(seq_len(nrow(g)), function(i) {
    if (g$type[i] == "N" && g$x_start[i] - min(g$x_start) < 1000)
      text(g[i,"x_start"], g[i,"y_start"], labels = parse(text = g[i,"degree_label"]),
           srt = g$angle_start[i], pos = 2, cex = .7)
    if (g$type[i] == "E" && g$y_start[i] - min(g$y_start) < 1000)
      text(g[i,"x_start"], g[i,"y_start"], labels = parse(text = g[i,"degree_label"]),
           srt = g$angle_start[i] - 90, pos = 1, cex = .7)
    if (g$type[i] == "N" && g$x_end[i] - max(g$x_end) > -1000)
      text(g[i,"x_end"], g[i,"y_end"], labels = parse(text = g[i,"degree_label"]),
           srt = g$angle_end[i], pos = 4, cex = .7)
    if (g$type[i] == "E" && g$y_end[i] - max(g$y_end) > -1000)
      text(g[i,"x_end"], g[i,"y_end"], labels = parse(text = g[i,"degree_label"]),
           srt = g$angle_end[i] - 90, pos = 3, cex = .7)
  }))
  plot(usa, graticule = st_crs(4326), axes = TRUE, lon = seq(-60,-130,by=-10))
}
```

---

`st_is`  test equality between the geometry type and a class or set of classes

**Description**

test equality between the geometry type and a class or set of classes
**Usage**

\[
\text{st\_is}(x, \text{type})
\]

**Arguments**

- **x**  
  object of class \text{sf}, \text{sfc} or \text{sfg}

- **type**  
  character; class, or set of classes, to test against

**Examples**

\[
\begin{align*}
\text{st\_is(st\_point(0:1), "POINT")} \\
\text{sfc = st\_sfc(st\_point(0:1), st\_linestring(matrix(1:6,,2))}) \\
\text{st\_is(sfc, "POINT")} \\
\text{st\_is(sfc, "POLYGON")} \\
\text{st\_is(sfc, "LINESTRING")} \\
\text{st\_is(st\_sf(a = 1:2, sfc), "LINESTRING")} \\
\text{st\_is(sfc, c("POINT", "LINESTRING")}
\end{align*}
\]

---

**st\_is\_longlat**  
**Assert whether simple feature coordinates are longlat degrees**

**Description**

Assert whether simple feature coordinates are longlat degrees

**Usage**

\[
\text{st\_is\_longlat}(x)
\]

**Arguments**

- **x**  
  object of class \text{sf} or \text{sfc}, or otherwise an object of a class that has an \text{st\_crs} method returning a \text{crs} object

**Value**

TRUE if \( x \) has geographic coordinates, FALSE if it has projected coordinates, or NA if is.na(st\_crs(x)).
Description

jitter geometries

Usage

st_jitter(x, amount, factor = 0.002)

Arguments

x
object of class sf or sfc

amount
numeric; amount of jittering applied; if missing, the amount is set to factor * the bounding box diagonal; units of coordinates.

factor
numeric; fractional amount of jittering to be applied

Details

jitters coordinates with an amount such that runif(1, -amount, amount) is added to the coordinates. x- and y-coordinates are jittered independently but all coordinates of a single geometry are jittered with the same amount, meaning that the geometry shape does not change. For longlat data, a latitude correction is made such that jittering in East and North directions are identical in distance in the center of the bounding box of x.

Examples

nc = st_read(system.file("gpkg/nc.gpkg", package="sf"))
pts = st_centroid(st_geometry(nc))
plot(pts)
plot(st_jitter(pts, .05), add = TRUE, col = 'red')
plot(st_geometry(nc))
plot(st_jitter(st_geometry(nc), factor = .01), add = TRUE, col = '#ff8888')

Description

spatial join, spatial filter
Usage

st_join(x, y, join, ...)

## S3 method for class 'sf'
st_join(
  x,
  y,
  join = st_intersects,
  ...,  
  suffix = c(".x", ".y"),
  left = TRUE,
  largest = FALSE
)

st_filter(x, y, ...)

## S3 method for class 'sf'
st_filter(x, y, ..., .predicate = st_intersects)

Arguments

x

object of class sf

y

object of class sf

join

geometry predicate function with the same profile as st_intersects; see details

...

for st_join: arguments passed on to the join function or to st_intersection when largest is TRUE; for st_filter arguments passed on to the .predicate function, e.g. prepared, or a pattern for st_relate

suffix

length 2 character vector; see merge

left

logical; if TRUE return the left join, otherwise an inner join; see details. see also left_join

largest

logical; if TRUE, return x features augmented with the fields of y that have the largest overlap with each of the features of x; see https://github.com/r-spatial/sf/issues/578

.predicate

geometry predicate function with the same profile as st_intersects; see details

Details

alternative values for argument join are:

- st.contains_properly
- st.contains
- st.covered_by
- st.covers
- st.crosses
- st.disjoint
- st.equals_exact
A left join returns all records of the x object with y fields for non-matched records filled with NA values; an inner join returns only records that spatially match.

To replicate the results of `st_within(x, y)` you will need to use `st_join(x, y, join = "st_within", left = FALSE).

Value

an object of class sf, joined based on geometry

Examples

```r
a = st_sf(a = 1:3,
  geom = st_sfc(st_point(c(1,1)), st_point(c(2,2)), st_point(c(3,3))))
b = st_sf(a = 11:14,
  geom = st_sfc(st_point(c(10,10)), st_point(c(2,2)), st_point(c(2,2)), st_point(c(3,3))))
st_join(a, b)
st_join(a, b, left = FALSE)

# two ways to aggregate y's attribute values outcome over x's geometries:
st_join(a, b) %>% aggregate(list(.$a.x), mean)
if (require(dplyr, quietly = TRUE)) {
  st_join(a, b) %>% group_by(a.x) %>% summarise(mean(a.y))
}

# example of largest = TRUE:
nc <- st_transform(st_read(system.file("shape/nc.shp", package="sf"), 2264)
gr = st_sf(
  label = apply(expand.grid(1:10, LETTERS[10:1])[,2:1], 1, paste0, collapse = " ",
  geom = st_make_grid(st_as_sfc(st_bbox(nc))))
gr$col = sf.colors(10, categorical = TRUE, alpha = .3)
# cut, to check, NA's work out:
gr = gr[-(1:30),]
nc_j <- st_join(nc, gr, largest = TRUE)
# the two datasets:
opar = par(mfrow = c(2,1), mar = rep(0,4))
plot(st_geometry(nc_j))
plot(st_geometry(gr), add = TRUE, col = gr$col)
text(st_coordinates(st_centroid(gr)), labels = gr$label)

# the joined dataset:
plot(st_geometry(nc_j), border = 'black', col = nc_j$col)
text(st_coordinates(st_centroid(nc_j)), labels = nc_j$label, cex = .8)
plot(st_geometry(gr), border = 'green', add = TRUE)
par(opar)

# st_filter keeps the geometries in x where .predicate(x,y) returns any match in y for x
```
\texttt{st\_filter(a, b)}

\# for an anti-join, use the union of y
\texttt{st\_filter(a, st\_union(b), .predicate = st\_disjoint)}

---

\textbf{st\_layers} \quad \textit{Return properties of layers in a datasource}

**Description**

Return properties of layers in a datasource

**Usage**

\texttt{st\_layers(dsn, options = character(0), do\_count = FALSE)}

**Arguments**

- **dsn** data source name (interpretation varies by driver - for some drivers, dsn is a file name, but may also be a folder, or contain the name and access credentials of a database)
- **options** character; driver dependent dataset open options, multiple options supported.
- **do\_count** logical; if TRUE, count the features by reading them, even if their count is not reported by the driver

**Value**

list object of class \texttt{sf\_layers} with elements

- **name** name of the layer
- **geomtype** list with for each layer the geometry types
- **features** number of features (if reported; see do\_count)
- **fields** number of fields
- **crs** list with for each layer the crs object
st_line_sample  Sample points on a linear geometry

Description

Sample points on a linear geometry

Usage

st_line_sample(x, n, density, type = "regular", sample = NULL)

Arguments

x  
object of class sf, sfc or sfg

n  
integer; number of points to choose per geometry; if missing, n will be computed as round(density * st_length(geom)).

density  
numeric; density (points per distance unit) of the sampling, possibly a vector of length equal to the number of features (otherwise recycled); density may be of class units.

type  
character; indicate the sampling type, either "regular" or "random"

Examples

ls = st_sfc(st_linestring(rbind(c(0,0),c(0,1))),
            st_linestring(rbind(c(0,0),c(10,0))))
ls = st_sfc(st_linestring(rbind(c(0,0),c(0,1))),
            st_linestring(rbind(c(0,0),c(.1,0))), crs = 4326)
try(st_line_sample(ls, density = 1/1000))  # error
st_line_sample(st_transform(ls, 3857), n = 5)  # five points for each line
st_line_sample(st_transform(ls, 3857), n = c(1, 3))  # one and three points
st_line_sample(st_transform(ls, 3857), density = 1/1000)  # one per km
st_line_sample(st_transform(ls, 3857), density = c(1/1000, 1/10000))  # one per km, one per 10 km
st_line_sample(st_transform(ls, 3857), density = units::set_units(1, 1/km))  # one per km
# five equidistant points including start and end:
st_line_sample(st_transform(ls, 3857), sample = c(0, 0.25, 0.5, 0.75, 1))
st_make_grid

Create a regular tessellation over the bounding box of an sf or sfc object

Description

Create a square or hexagonal grid covering the bounding box of the geometry of an sf or sfc object

Usage

```r
st_make_grid(
  x,
  cellsize = c(diff(st_bbox(x)[c(1, 3)]), diff(st_bbox(x)[c(2, 4)]))/n,
  offset = st_bbox(x)[c("xmin", "ymin")],
  n = c(10, 10),
  crs = if (missing(x)) NA_crs else st_crs(x),
  what = "polygons",
  square = TRUE,
  flat_topped = FALSE
)
```

Arguments

- **x**: object of class sf or sfc
- **cellsize**: numeric of length 1 or 2 with target cellsize: for square or rectangular cells the width and height, for hexagonal cells the distance between opposite edges (edge length is cellsize/sqrt(3)). A length units object can be passed, or an area unit object with area size of the square or hexagonal cell.
- **offset**: numeric of length 2; lower left corner coordinates (x, y) of the grid
- **n**: integer of length 1 or 2, number of grid cells in x and y direction (columns, rows)
- **crs**: object of class crs; coordinate reference system of the target of the target grid in case argument x is missing, if x is not missing, its crs is inherited.
- **what**: character; one of: "polygons", "corners", or "centers"
- **square**: logical; if FALSE, create hexagonal grid
- **flat_topped**: logical; if TRUE generate flat topped hexagons, else generate pointy topped

Value

Object of class sfc (simple feature geometry list column) with, depending on what and square, square or hexagonal polygons, corner points of these polygons, or center points of these polygons.
Examples

plot(st_make_grid(what = "centers"), axes = TRUE)
plot(st_make_grid(what = "corners"), add = TRUE, col = 'green', pch=3)
sfc = st_sfc(st_polygon(list(rbind(c(0,0), c(1,0), c(1,1), c(0,0)))))
plot(st_make_grid(sfc, cellsize = .1, square = FALSE))
plot(sfc, add = TRUE)
# non-default offset:
plot(st_make_grid(sfc, cellsize = .1, square = FALSE, offset = c(0, .05 / (sqrt(3)/2))))
plot(sfc, add = TRUE)
nc = st_read(system.file("shape/nc.shp", package="sf"))
g = st_make_grid(nc)
plot(g)
plot(st_geometry(nc), add = TRUE)
# g[nc] selects cells that intersect with nc:
plot(g[nc], col = '#ff000088', add = TRUE)

---

**st_m_range**

Return 'm' range of a simple feature or simple feature set

Description

Return 'm' range of a simple feature or simple feature set

Usage

```r
## S3 method for class 'm_range'
is.na(x)

st_m_range(obj, ...)

## S3 method for class 'POINT'
st_m_range(obj, ...)

## S3 method for class 'MULTIPOINT'
st_m_range(obj, ...)

## S3 method for class 'LINESTRING'
st_m_range(obj, ...)

## S3 method for class 'POLYGON'
st_m_range(obj, ...)

## S3 method for class 'MULTILINESTRING'
st_m_range(obj, ...)

## S3 method for class 'MULTIPOLYGON'
st_m_range(obj, ...)
```

```
## S3 method for class 'GEOMETRYCOLLECTION'
st_m_range(obj, ...)

## S3 method for class 'MULTISURFACE'
st_m_range(obj, ...)

## S3 method for class 'MULTICURVE'
st_m_range(obj, ...)

## S3 method for class 'CURVEPOLYGON'
st_m_range(obj, ...)

## S3 method for class 'COMPOUNDCURVE'
st_m_range(obj, ...)

## S3 method for class 'POLYHEDRALSURFACE'
st_m_range(obj, ...)

## S3 method for class 'TIN'
st_m_range(obj, ...)

## S3 method for class 'TRIANGLE'
st_m_range(obj, ...)

## S3 method for class 'CIRCULARSTRING'
st_m_range(obj, ...)

## S3 method for class 'sfc'
st_m_range(obj, ...)

## S3 method for class 'sf'
st_m_range(obj, ...)

## S3 method for class 'numeric'
st_m_range(obj, ..., crs = NA_crs_)

NA_m_range_

### Arguments

- **x**: object of class m_range
- **obj**: object to compute the m range from
- **...**: ignored
- **crs**: object of class crs, or argument to st_crs, specifying the CRS of this bounding box.
Format

An object of class m_range of length 2.

Details

NA_m_range_ represents the missing value for a m_range object

Value

a numeric vector of length two, with mmin and mmax values; if obj is of class sf or sfc the object if obj is of class sf or sfc the object returned has a class m_range

Examples

a = st_sf(a = 1:2, geom = st_sfc(st_point(0:3), st_point(1:4)), crs = 4326)
st_m_range(a)
st_m_range(c(mmin = 16.1, mmax = 16.6), crs = st_crs(4326))

Description

get index of nearest feature

Usage

st_nearest_feature(
  x,
  y,
  ...,  
  check_crs = TRUE,
  longlat = isTRUE(st_is_longlat(x))
)

Arguments

x object of class sfg, sfc or sf
y object of class sfg, sfc or sf; if missing, features in x will be compared to all remaining features in x.
... ignored
check_crs logical; should x and y be checked for CRS equality?
longlat logical; does x have ellipsoidal coordinates?
Value

for each feature (geometry) in x the index of the nearest feature (geometry) in set y, or in the remaining set of x if y is missing; empty geometries result in NA indexes

See Also

st_nearest_points for finding the nearest points for pairs of feature geometries

Examples

```r
ls1 = st_linestring(rbind(c(0,0), c(1,0)))
ls2 = st_linestring(rbind(c(0,0.1), c(1,0.1)))
ls3 = st_linestring(rbind(c(0,1), c(1,1)))
(l = st_sfc(ls1, ls2, ls3))

p1 = st_point(c(0.1, -0.1))
p2 = st_point(c(0.1, 0.11))
p3 = st_point(c(0.1, 0.09))
p4 = st_point(c(0.1, 0.9))

(p = st_sfc(p1, p2, p3, p4))
try(st_nearest_feature(p, l))
try(st_nearest_points(p, l[st_nearest_feature(p,l)], pairwise = TRUE))

r = sqrt(2)/10
b1 = st_buffer(st_point(c(.1,.1)), r)
b2 = st_buffer(st_point(c(.9,.1)), r)
b3 = st_buffer(st_point(c(.9,.9)), r)
circles = st_sfc(b1, b2, b3)
plot(circles, col = NA, border = 2:4)
pts = st_sfc(st_point(c(.3,.1)), st_point(c(.6,.2)), st_point(c(.6,.6)), st_point(c(.4,.8)))
plot(pts, add = TRUE, col = 1)
# draw points to nearest circle:
nearest = try(st_nearest_feature(pts, circles))
if (inherits(nearest, "try-error")) # GEOS 3.6.1 not available
    nearest = c(1, 3, 2, 2)
ls = st_nearest_points(pts, circles[nearest], pairwise = TRUE)
plot(ls, col = 5:8, add = TRUE)
# compute distance between pairs of nearest features:
st_distance(pts, circles[nearest], by_element = TRUE)
```

---

**st_nearest_points**

get nearest points between pairs of geometries

Description

get nearest points between pairs of geometries
Usage

```
st_nearest_points(x, y, ...)  
```

```
## S3 method for class 'sfc'
st_nearest_points(x, y, ..., pairwise = FALSE)
```

```
## S3 method for class 'sfg'
st_nearest_points(x, y, ...)
```

```
## S3 method for class 'sf'
st_nearest_points(x, y, ...)
```

Arguments

- `x` object of class `sfg`, `sfc` or `sf`
- `y` object of class `sfg`, `sfc` or `sf`
- `...` ignored
- `pairwise` logical; if `FALSE` (default) return nearest points between all pairs, if `TRUE`, return nearest points between subsequent pairs.

Details

In case `x` lies inside `y`, when using S2, the end points are on polygon boundaries, when using GEOS the end point are identical to `x`.

Value

An `sfc` object with all two-point LINESTRING geometries of point pairs from the first to the second geometry, of length `x * y`, with `y` cycling fastest. See examples for ideas how to convert these to POINT geometries.

See Also

- `st_nearest_feature` for finding the nearest feature

Examples

```
r = sqrt(2)/10
pt1 = st_point(c(.1,.1))
pt2 = st_point(c(.9,.9))
pt3 = st_point(c(.9,.1))
b1 = st_buffer(pt1, r)
b2 = st_buffer(pt2, r)
b3 = st_buffer(pt3, r)
(ls0 = st_nearest_points(b1, b2)) # sfg
(ls = st_nearest_points(st_sfc(b1), st_sfc(b2, b3))) # sfc
plot(b1, xlim = c(-.2,1.2), ylim = c(-.2,1.2), col = NA, border = 'green')
plot(st_sfc(b2, b3), add = TRUE, col = NA, border = 'blue')
plot(ls, add = TRUE, col = 'red')
```
nc = st_read(system.file("gpkg/nc.gpkg", package="sf"))
plot(st_geometry(nc))
ls = st_nearest_points(nc[1,], nc)
plot(ls, col = 'red', add = TRUE)
pts = st_cast(ls, "POINT") # gives all start & end points
# starting, "from" points, corresponding to x:
plot(pts[seq(1, 200, 2)], add = TRUE, col = 'blue')
# ending, "to" points, corresponding to y:
plot(pts[seq(2, 200, 2)], add = TRUE, col = 'green')

---

**st_normalize**

*Normalize simple features*

**Description**

`st_normalize` transforms the coordinates in the input feature to fall between 0 and 1. By default the current domain is set to the bounding box of the input, but other domains can be used as well.

**Usage**

```r
st_normalize(x, domain = st_bbox(x), ...)
```

**Arguments**

- `x` object of class sf, sfc or sfg
- `domain` The domain `x` should be normalized from as a length 4 vector of the form `c(xmin, ymin, xmax, ymax)`. Defaults to the bounding box of `x`
- `...` ignored

**Examples**

```r
p1 = st_point(c(7,52))
st_normalize(p1, domain = c(0, 0, 10, 100))

p2 = st_point(c(-30,20))
sfc = st_sfc(p1, p2, crs = 4326)
sfc
sfc_norm <- st_normalize(sfc)
st_bbox(sfc_norm)
```
Description

Get precision
Set precision

Usage

\texttt{st\_precision(x)}

\texttt{st\_set\_precision(x, \text{precision})}

\texttt{st\_precision(x) \leftarrow \text{value}}

Arguments

\begin{itemize}
\item \textbf{x} \hspace{1cm} object of class \texttt{sfc} or \texttt{sf}
\item \textbf{precision} \hspace{1cm} numeric, or object of class \texttt{units} with distance units (but see details); see \texttt{st\_as\_binary} for how to do this.
\item \textbf{value} \hspace{1cm} precision value
\end{itemize}

Details

If \texttt{precision} is a \texttt{units} object, the object on which we set precision must have a coordinate reference system with compatible distance units.

Setting a precision has no direct effect on coordinates of geometries, but merely set an attribute tag to an \texttt{sfc} object. The effect takes place in \texttt{st\_as\_binary} or, more precise, in the C++ function \texttt{CPL\_write\_wkb}, where simple feature geometries are being serialized to well-known-binary (WKB). This happens always when routines are called in GEOS library (geometrical operations or predicates), for writing geometries using \texttt{st\_write} or \texttt{write\_sf}, \texttt{st\_make\_valid} in package \texttt{lwgeom}; also \texttt{aggregate} and \texttt{summarise} by default union geometries, which calls a GEOS library function. Routines in these libraries receive rounded coordinates, and possibly return results based on them. \texttt{st\_as\_binary} contains an example of a roundtrip of \texttt{sfc} geometries through WKB, in order to see the rounding happening to R data.

The reason to support precision is that geometrical operations in GEOS or liblwgeom may work better at reduced precision. For writing data from R to external resources it is harder to think of a good reason to limiting precision.

See Also

\texttt{st\_as\_binary} for an explanation of what setting precision does, and the examples therein.
Examples

```r
x <- st_sfc(st_point(c(pi, pi)))
st_precision(x)
st_precision(x) <- 0.01
st_precision(x)
```

---

**st_read**

Read simple features or layers from file or database

### Description

Read simple features from file or database, or retrieve layer names and their geometry type(s)

Read PostGIS table directly through DBI and RPostgreSQL interface, converting Well-Know Binary geometries to sfc

### Usage

```r
st_read(dsn, layer, ...)
```

#### S3 method for class 'character'

```r
st_read(
  dsn,
  layer,
  ..., 
  query = NA,
  options = NULL,
  quiet = FALSE,
  geometry_column = 1L,
  type = 0,
  promote_to_multi = TRUE,
  stringsAsFactors = sf_stringsAsFactors(),
  int64_as_string = FALSE,
  check_ring_dir = FALSE,
  fid_column_name = character(0),
  drivers = character(0),
  wkt_filter = character(0),
  optional = FALSE
)
```

```r
read_sf(..., quiet = TRUE, stringsAsFactors = FALSE, as_tibble = TRUE)
```

#### S3 method for class 'DBIObject'

```r
st_read(
  dsn = NULL,
  layer = NULL,
  query = NULL,
  ...) 
```

EWKB = TRUE,
quiet = TRUE,
as_tibble = FALSE,
geometry_column = NULL,
)

Arguments

dsn                  data source name (interpretation varies by driver - for some drivers, dsn is a file name, but may also be a folder, or contain the name and access credentials of a database); in case of GeoJSON, dsn may be the character string holding the geojson data. It can also be an open database connection.
layer                layer name (varies by driver, may be a file name without extension); in case layer is missing, st_read will read the first layer of dsn, give a warning and (unless quiet = TRUE) print a message when there are multiple layers, or give an error if there are no layers in dsn. If dsn is a database connection, then layer can be a table name or a database identifier (see Id). It is also possible to omit layer and rather use the query argument.

query               SQL query to select records; see details
options             character; driver dependent dataset open options, multiple options supported. For possible values, see the "Open options" section of the GDAL documentation of the corresponding driver, and https://github.com/r-spatial/sf/issues/1157 for an example.
quiet               logical; suppress info on name, driver, size and spatial reference, or signaling no or multiple layers
geometry_column     integer or character; in case of multiple geometry fields, which one to take?
type                integer; ISO number of desired simple feature type; see details. If left zero, and promote_to_multi is TRUE, in case of mixed feature geometry types, conversion to the highest numeric type value found will be attempted. A vector with different values for each geometry column can be given.
promote_to_multi     logical; in case of a mix of Point and MultiPoint, or of LineString and MultiLineString, or of Polygon and MultiPolygon, convert all to the Multi variety; defaults to TRUE
stringsAsFactors     logical; logical: should character vectors be converted to factors? Default for read_sf or R version >= 4.1.0 is FALSE, for st_read and R version < 4.1.0 equal to default.stringsAsFactors()
int64_as_string      logical; if TRUE, Int64 attributes are returned as string; if FALSE, they are returned as double and a warning is given when precision is lost (i.e., values are larger than 2^53).
check_ring_dir       logical; if TRUE, polygon ring directions are checked and if necessary corrected (when seen from above: exterior ring counter clockwise, holes clockwise)
st_read

**Parameters**

- `fid_column_name` character; name of column to write feature IDs to; defaults to not doing this
- `drivers` character; limited set of driver short names to be tried (default: try all)
- `wkt_filter` character; WKT representation of a spatial filter (may be used as bounding box, selecting overlapping geometries); see examples
- `optional` logical; passed to `as.data.frame`; always TRUE when `as_tibble` is TRUE
- `as_tibble` logical; should the returned table be of class tibble or data.frame?
- `EWKB` logical; is the WKB of type EWKB? if missing, defaults to TRUE

**Details**

For `geometry_column`, see also [https://trac.osgeo.org/gdal/wiki/rfc41_multiple_geometry_fields](https://trac.osgeo.org/gdal/wiki/rfc41_multiple_geometry_fields)

For values for `type` see [https://en.wikipedia.org/wiki/Well-known_text#Well-known_binary](https://en.wikipedia.org/wiki/Well-known_text#Well-known_binary), but note that not every target value may lead to successful conversion. The typical conversion from POLYGON (3) to MULTIPOLYGON (6) should work; the other way around (type=3), secondary rings from MULTIPOLYGONS may be dropped without warnings. `promote_to_multi` is handled on a per-geometry column basis; `type` may be specified for each geometry column.

Note that stray files in data source directories (such as `*.dbf`) may lead to spurious errors that accompanying `*.shp` are missing.

In case of problems reading shapefiles from USB drives on OSX, please see [https://github.com/r-spatial/sf/issues/252](https://github.com/r-spatial/sf/issues/252).

For query with a character `dsn` the query text is handed to ‘ExecuteSQL’ on the GDAL/OGR data set and will result in the creation of a new layer (and `layer` is ignored). See ‘OGRSQL’ [https://gdal.org/user/ogr_sql_dialect.html](https://gdal.org/user/ogr_sql_dialect.html) for details. Please note that the ‘FID’ special field is driver-dependent, and may be either 0-based (e.g. ESRI Shapefile), 1-based (e.g. MapInfo) or arbitrary (e.g. OSM). Other features of OGRSQL are also likely to be driver dependent. The available layer names may be obtained with `st_layers`. Care will be required to properly escape the use of some layer names.

`read_sf` and `write_sf` are aliases for `st_read` and `st_write`, respectively, with some modified default arguments. `read_sf` and `write_sf` are quiet by default: they do not print information about the data source. `read_sf` returns an sf-tibble rather than an sf-data.frame. `write_sf` delete layers by default: it overwrites existing files without asking or warning.

If `table` is not given but `query` is, the spatial reference system (crs) of the table queried is only available in case it has been stored into each geometry record (e.g., by PostGIS, when using EWKB)

The function will automatically find the ‘geometry‘ type columns for drivers that support it. For the other drivers, it will try to cast all the character columns, which can be slow for very wide tables.

**Value**

object of class `sf` when a layer was successfully read; in case argument `layer` is missing and data source `dsn` does not contain a single layer, an object of class `sf_layers` is returned with the layer names, each with their geometry type(s). Note that the number of layers may also be zero.
Note

The use of \texttt{system.file} in examples make sure that examples run regardless where R is installed: typical users will not use \texttt{system.file} but give the file name directly, either with full path or relative to the current working directory (see \texttt{getwd}). "Shapefiles" consist of several files with the same basename that reside in the same directory, only one of them having extension \texttt{.shp}.

See Also

\texttt{st_layers, st_drivers}

Examples

\begin{verbatim}
nc = st_read(system.file("shape/nc.shp", package="sf"))
summary(nc) \# note that \texttt{AREA} was computed using Euclidian area on lon/lat degrees

\texttt{## only three fields by select clause}
\texttt{## only two features by where clause}
\texttt{nc_sql = st_read(system.file("shape/nc.shp", package="sf"),
\texttt{query = "SELECT NAME, SID74, FIPS FROM \"nc\" WHERE BIR74 > 20000")}

\texttt{## Not run:}
library(sp)
example(meuse, ask = FALSE, echo = FALSE)
try(st_write(st_as_sf(meuse), "PG:dbname=postgis", "meuse",
layer_options = "OVERWRITE=true"))
try(st_meuse <- st_read("PG:dbname=postgis", "meuse"))
if (exists("st_meuse"))
  summary(st_meuse)

\texttt{## End(Not run)}

\texttt{## Not run:}
\texttt{## note that we need special escaping of layer within single quotes (nc.gpkg)}
\texttt{## and that geom needs to be included in the select, otherwise we don't detect it}
layer <- st_layers(system.file("gpkg/nc.gpkg", package = "sf"))$name[1]
nc_gpkg_sql = st_read(system.file("gpkg/nc.gpkg", package = "sf"),
query = sprintf("SELECT NAME, SID74, FIPS, geom FROM \"%s\" WHERE BIR74 > 20000", layer))

\texttt{## End(Not run)}
\texttt{# spatial filter, as wkt:}
wkt = st_as_text(st_geometry(nc[1,]))
\texttt{# filter by (bbox overlaps of) first feature geometry:}
st_read(system.file("gpkg/nc.gpkg", package="sf"), wkt_filter = wkt)
\texttt{# read geojson from string:}
geojson_txt <- paste("\{"type":"MultiPoint","coordinates":",
           \[\[3.2,4],[3,4.6],[3.8,4.4],[3.5,3.8],[3.4,3.6],[3.9,4.5]\],\]*)
x = st_read(geojson_txt)
x
\texttt{## Not run:}
library(RPostgreSQL)
try(conn <- dbConnect(PostgreSQL(), dbname = "postgis"))
if (exists("conn") && inherits(conn, "try-error")) {
  x = st_read(conn, "meuse", query = "select * from meuse limit 3;")
\end{verbatim}
st_relate

```
x = st_read(conn, table = "public.meuse")
print(st_crs(x)) # SRID resolved by the database, not by GDAL!
dbDisconnect(conn)
```

```
## End(Not run)
```

---

**st_relate**

Compute DE9-IM relation between pairs of geometries, or match it to a given pattern

**Description**

Compute DE9-IM relation between pairs of geometries, or match it to a given pattern

**Usage**

```
st_relate(x, y, pattern = NA_character_, sparse = !is.na(pattern))
```

**Arguments**

- **x**: object of class sf, sfc or sfg
- **y**: object of class sf, sfc or sfg
- **pattern**: character; define the pattern to match to, see details.
- **sparse**: logical; should a sparse matrix be returned (TRUE) or a dense matrix?

**Value**

In case **pattern** is not given, **st_relate** returns a dense character matrix; element [i,j] has nine characters, referring to the DE9-IM relationship between x[i] and y[j], encoded as IxIy,IxBy,IxEy,BxIy,BxBy,BxEy,ExIy,ExBy,ExEy where I refers to interior, B to boundary, and E to exterior, and e.g. BxIy the dimensionality of the intersection of the boundary of x[i] and the interior of y[j], which is one of 0,1,2,F, digits denoting dimensionality, F denoting not intersecting. When **pattern** is given, a dense logical matrix or sparse index list returned with matches to the given pattern; see **st_intersection** for a description of the returned matrix or list. See also [DE-9IM](https://en.wikipedia.org/wiki/DE-9IM) for further explanation.

**Examples**

```
p1 = st_point(c(0,0))
p2 = st_point(c(2,2))
pol1 = st_polygon(list(rbind(c(0,0),c(1,0),c(1,1),c(0,1),c(0,0)))) - 0.5
pol2 = pol1 + 1
pol3 = pol1 + 2
st_relate(st_sfc(p1, p2), st_sfc(pol1, pol2, pol3))
sfc = st_sfc(st_point(c(0,0)), st_point(c(3,3)))
grd = st_make_grid(sfc, n = c(3,3))
st_intersects(grd)
```
st_relate(grd, pattern = "****1****") # sides, not corners, internals
st_relate(grd, pattern = "****0****") # only corners touch
st_rook = function(a, b = a) st_relate(a, b, pattern = "F***1****")
st_rook(grd)
# queen neighbours, see \url{https://github.com/r-spatial/sf/issues/234#issuecomment-300511129}
st_queen <- function(a, b = a) st_relate(a, b, pattern = "F***T****")

---

**Description**

Sample points on or in (sets of) spatial features. By default, returns a pre-specified number of points that is equal to size (if type = "random" and exact = TRUE) or an approximation of size otherwise. spatstat methods are interfaced and do not use the size argument, see examples.

**Usage**

```r
st_sample(x, size, ...)
```

## S3 method for class 'sf'

```r
st_sample(x, size, ...)
```

## S3 method for class 'sfc'

```r
st_sample(
    x,
    size,
    ...,
    type = "random",
    exact = TRUE,
    warn_if_not_integer = TRUE,
    by_polygon = FALSE,
    progress = FALSE
)
```

## S3 method for class 'sfg'

```r
st_sample(x, size, ...)
```

**Arguments**

- `x` object of class `sf` or `sfc`
- `size` sample size(s) requested; either total size, or a numeric vector with sample sizes for each feature geometry. When sampling polygons, the returned sampling size may differ from the requested size, as the bounding box is sampled, and sampled points intersecting the polygon are returned.
- `...` passed on to `sample` for multipoint sampling, or to spatstat functions for spatstat sampling types (see details)
type character; indicates the spatial sampling type; one of random, hexagonal (triangular really), regular, Fibonacci, or one of the spatstat methods such as Thomas for calling spatstat.random::rThomas (see Details).

exact logical; should the length of output be exactly

warn_if_not_integer logical; if FALSE then no warning is emitted if size is not an integer

by_polygon logical; for MULTIPOLYGON geometries, should the effort be split by POLYGON? See https://github.com/r-spatial/sf/issues/1480 the same as specified by size? TRUE by default. Only applies to polygons, and when type = "random".

progress logical; if TRUE show progress bar (only if size is a vector).

Details

The function is vectorised: it samples size points across all geometries in the object if size is a single number, or the specified number of points in each feature if size is a vector of integers equal in length to the geometry of x.

if x has dimension 2 (polygons) and geographical coordinates (long/lat), uniform random sampling on the sphere is applied, see e.g. http://mathworld.wolfram.com/SpherePointPicking.html.

For regular or hexagonal sampling of polygons, the resulting size is only an approximation.

As parameter called offset can be passed to control ("fix") regular or hexagonal sampling: for polygons a length 2 numeric vector (by default: a random point from st_bbox(x)); for lines use a number like runif(1).


For regular sampling on the sphere, see also geosphere::regularCoordinates.

Sampling methods from package spatstat are interfaced (see examples), and need their own parameters to be set. For instance, to use spatstat.random::rThomas(), set type = "Thomas".

Value

an sfc object containing the sampled POINT geometries

Examples

nc = st_read(system.file("shape/nc.shp", package="sf"))
p1 = st_sample(nc[1:3, ], 6)
p2 = st_sample(nc[1:3, ], 1:3)
plot(st_geometry(nc)[1:3])
plot(p1, add = TRUE)
plot(p2, add = TRUE, pch = 2)
x = st_sfc(st_polygon(list(rbind(c(0,0),c(90,0),c(90,90),c(0,90),c(0,0)))), crs = st_crs(4326))
plot(x, axes = TRUE, graticule = TRUE)
if (sf_extSoftVersion()["proj.4"] >= "4.9.0")
  plot(p <- st_sample(x, 1000), add = TRUE)
if (require(lwgeom, quietly = TRUE)) { # for st_segmentize()
x2 = st_transform(st_segmentize(x, 1e4), st_crs("+proj=ortho +lat_0=30 +lon_0=45"))
g = st_transform(st_graticule(), st_crs("+proj=ortho +lat_0=30 +lon_0=45"))}
st_shift_longitude

Shift or re-center geographical coordinates for a Pacific view

Description

All longitudes < 0 are added to 360, to avoid for instance parts of Alaska being represented on the far left and right of a plot because they have values straddling 180 degrees. In general, using a projected coordinate reference system is to be preferred, but this method permits a geographical coordinate reference system to be used. This is the sf equivalent of `recenter` in the sp package and `ST_ShiftLongitude` in PostGIS.
Usage

st_shift_longitude(x)

## S3 method for class 'sfc'
st_shift_longitude(x, ...)

## S3 method for class 'sf'
st_shift_longitude(x, ...)

Arguments

x          object of class sf or sfc
...        ignored

Examples

## sfc
pt1 = st_point(c(-170, 50))
pt2 = st_point(c(170, 50))
(sfc = st_sfc(pt1, pt2))
sfc = st_set_crs(sfc, 4326)
st_shift_longitude(sfc)

## sf
d = st_as_sf(data.frame(id = 1:2, geometry = sfc))
st_shift_longitude(d)

---

**st_transform**  
*Transform or convert coordinates of simple feature*

Description

Transform or convert coordinates of simple feature

Usage

st_can_transform(src, dst)
st_transform(x, crs, ...)

## S3 method for class 'sfc'
st_transform(
x,
crs = st_crs(x),
...,
aoi = numeric(0),
pipeline = character(0),
...,
...,
reverse = FALSE,
desired_accuracy = -1,
allow_ballpark = TRUE,
partial = TRUE,
check = FALSE
)
## S3 method for class 'sf'
st_transform(x, crs = st_crs(x), ...)

## S3 method for class 'sfg'
st_transform(x, crs = st_crs(x), ...)

st_wrap_dateline(x, options, quiet)
## S3 method for class 'sfc'
st_wrap_dateline(x, options = "WRAPDATELINE=YES", quiet = TRUE)

## S3 method for class 'sf'
st_wrap_dateline(x, options = "WRAPDATELINE=YES", quiet = TRUE)

## S3 method for class 'sfg'
st_wrap_dateline(x, options = "WRAPDATELINE=YES", quiet = TRUE)

sf_proj_info(type = "proj", path)

Arguments

src source crs
dst destination crs
x object of class sf, sfc or sfg
crs target coordinate reference system: object of class 'crs', or input string for st_crs
... ignored
aoi area of interest, in degrees: WestLongitude, SouthLatitude, EastLongitude, NorthLatitude
pipeline character; coordinate operation pipeline, for overriding the default operation
reverse boolean; has only an effect when pipeline is defined: if TRUE, the inverse operation of the pipeline is applied
desired_accuracy numeric; Only coordinate operations that offer an accuracy of at least the one specified will be considered; a negative value disables this feature (requires GDAL >= 3.3)
allow_ballpark logical; are ballpark (low accuracy) transformations allowed? (requires GDAL >= 3.3)
partial logical; allow for partial projection, if not all points of a geometry can be projected (corresponds to setting environment variable OGR_ENABLE_PARTIAL_REPROJECTION to TRUE)
check logical; if TRUE, perform a sanity check on resulting polygons
options character; should have "WRAPDATELINE=YES" to function; another parameter that is used is "DATELINEOFFSET=10" (where 10 is the default value)
quiet logical; print options after they have been parsed?
type character; one of have_datum_files, proj, ellps, datum, units or prime меридианы; see Details.
path character; PROJ search path to be set

Details

st_can_transform returns a boolean indicating whether coordinates with CRS src can be transformed into CRS dst

Transforms coordinates of object to new projection. Features that cannot be transformed are returned as empty geometries. Transforms using the pipeline= argument may fail if there is ambiguity in the axis order of the specified coordinate reference system; if you need the traditional GIS order, use "OGC:CRS84", not "EPSG:4326". Extra care is needed with the ESRI Shapefile format, because WKT1 does not store axis order unambiguously.

The st_transform method for sfg objects assumes that the CRS of the object is available as an attribute of that name.


sf_proj_info lists the available projections, ellipses, datums, units, or data search path of the PROJ library when type is equal to proj, ellps, datum, units or path; when type equals have_datum_files a boolean is returned indicating whether datum files are installed and accessible (checking for conus).

for PROJ >= 6, sf_proj_info does not provide option type = "datums". PROJ < 6 does not provide the option type = "prime меридианы".

for PROJ >= 7.1.0, the "units" query of sf_proj_info returns the to_meter variable as numeric, previous versions return a character vector containing a numeric expression.

See Also

st_transform_proj, part of package lwgeom.
sf_project projects a matrix of coordinates, bypassing GDAL altogether
st_break_antimeridian

Examples

```r
  p1 = st_point(c(7,52))
  p2 = st_point(c(-30,20))
  sfc = st_sfc(p1, p2, crs = 4326)
  sfc
  st_transform(sfc, 3857)
  st_transform(st_sf(a=2:1, geom=sfc), "+init=epsg:3857")
  try(st_transform(sfc, 3857, aoi = c(-280,-90,180,90)))
  if (sf_extSoftVersion()["GDAL"] >= "3.0.0") {
```
st_viewport

Create viewport from sf, sfc or sfg object

Description

Create viewport from sf, sfc or sfg object

Usage

st_viewport(x, ..., bbox = st_bbox(x), asp)

Arguments

x object of class sf, sfc or sfg object
...
bbox the bounding box used for aspect ratio
asp numeric; target aspect ratio (y/x), see Details

Details

parameters width, height, xscale and yscale are set such that aspect ratio is honoured and plot size is maximized in the current viewport; others can be passed as ...

If asp is missing, it is taken as 1, except when isTRUE(st_is_longlat(x)), in which case it is set to 1.0 /cos(y), with y the middle of the latitude bounding box.

Value

The output of the call to viewport
Examples

```r
library(grid)
nc = st_read(system.file("shape/nc.shp", package="sf"))
grid.newpage()
pushViewport(viewport(width = 0.8, height = 0.8))
pushViewport(st_viewport(nc))
invisible(lapply(st_geometry(nc), function(x) grid.draw(st_as_grob(x, gp = gpar(fill = "red")))))
```

---

**st_write**  
Write simple features object to file or database

### Description
Write simple features object to file or database

### Usage

```r
st_write(obj, dsn, layer, ...)

## S3 method for class 'sfc'
st_write(obj, dsn, layer, ...)

## S3 method for class 'sf'
st_write(
  obj,
  dsn,
  layer = NULL,
  ...,
  driver = guess_driver_can_write(dsn),
  dataset_options = NULL,
  layer_options = NULL,
  quiet = FALSE,
  factorsAsCharacter = TRUE,
  append = NA,
  delete_dsn = FALSE,
  delete_layer = !is.na(append) & & !append,
  fid_column_name = NULL,
  config_options = character(0)
)

## S3 method for class 'data.frame'
st_write(obj, dsn, layer = NULL, ...)

write_sf(..., quiet = TRUE, append = FALSE, delete_layer = !append)
```

### st_delete

```r
st_delete(dsn,
```
layer = character(0),
driver = guess_driver_can_write(dsn),
quiet = FALSE
)

Arguments

obj object of class sf or sfc
dsn data source name. Interpretation varies by driver: can be a filename, a folder, a
database name, or a Database Connection (we officially test support for RPostgres::Postgres() connections).
layer layer name. Varies by driver, may be a file name without extension; for database connection, it is the name of the table. If layer is missing, the basename of dsn is taken.
... other arguments passed to dbWriteTable when dsn is a Database Connection
driver character; name of driver to be used; if missing and dsn is not a Database Connection, a driver name is guessed from dsn; st_drivers() returns the drivers that are available with their properties; links to full driver documentation are found at https://gdal.org/drivers/vector/index.html
dataset_options character; driver dependent dataset creation options; multiple options supported.
layer_options character; driver dependent layer creation options; multiple options supported.
quiet logical; suppress info on name, driver, size and spatial reference
factorsAsCharacter logical; convert factor levels to character strings (TRUE, default), otherwise into
numbers when factorsAsCharacter is FALSE. For database connections, factorsAsCharacter is always TRUE.
append logical; should we append to an existing layer, or replace it? if TRUE append, if FALSE replace. The default for st_write is NA which raises an error if the layer exists. The default for write_sf is FALSE, which overwrites any existing data. See also next two arguments for more control on overwrite behavior.
delete_dsn logical; delete data source dsn before attempting to write?
delete_layer logical; delete layer layer before attempting to write? The default for st_write is FALSE which raises an error if the layer exists. The default for write_sf is TRUE.
fid_column_name character, name of column with feature IDs; if specified, this column is no longer written as feature attribute.
config_options character, named vector with GDAL config options

Details

Columns (variables) of a class not supported are dropped with a warning.

When updating an existing layer, records are appended to it if the updating object has the right variable names and types. If names don’t match an error is raised. If types don’t match, behaviour is undefined: GDAL may raise warnings or errors or fail silently.
When deleting layers or data sources is not successful, no error is emitted. `delete_dsn` and `delete_layer` should be handled with care; the former may erase complete directories or databases. `st_delete` deletes layer(s) in a data source, or a data source if layers are omitted; it returns TRUE on success, FALSE on failure, invisibly.

**Value**

`obj`, invisibly

**See Also**

`st_drivers`, `dbWriteTable`

**Examples**

```r
nc = st_read(system.file("shape/nc.shp", package="sf"))
st_write(nc, paste0(tempdir(), "/", "nc.shp"))
st_write(nc, paste0(tempdir(), "/", "nc.shp"), delete_layer = TRUE) # overwrites
if (require(sp, quietly = TRUE)) {
  data(meuse, package = "sp") # loads data.frame from sp
  meuse_sf = st_as_sf(meuse, coords = c("x", "y"), crs = 28992)
  # writes X and Y as columns:
  st_write(meuse_sf, paste0(tempdir(), "/", "meuse.csv"), layer_options = "GEOMETRY=AS_XY")
  st_write(meuse_sf, paste0(tempdir(), "/", "meuse.csv"), layer_options = "GEOMETRY=AS_WKT",
           delete_dsn=TRUE) # overwrites
  ## Not run:
  library(sp)
  example(meuse, ask = FALSE, echo = FALSE)
  try(st_write(st_as_sf(meuse), "PG:dbname=postgis", "meuse_sf",
               layer_options = c("OVERWRITE=yes", "LAUNDER=true")))
  demo(nc, ask = FALSE)
  try(st_write(nc, "PG:dbname=postgis", "sids", layer_options = "OVERWRITE=true"))
  ## End(Not run)
}
```

---

**st_zm**  
*Drop or add Z and/or M dimensions from feature geometries*

**Description**

Drop Z and/or M dimensions from feature geometries, resetting classes appropriately

**Usage**

```
st_zm(x, ..., drop = TRUE, what = "ZM")
```
Arguments

- **x**: object of class sfg, sfc or sf
- **drop**: logical; drop, or (FALSE) add?
- **what**: character which dimensions to drop or add

Details

Only combinations drop=TRUE, what = "ZM", and drop=FALSE, what="Z" are supported so far. In case add=TRUE, x should have XY geometry, and zero values are added for Z.

Examples

```r
st_zm(st_linestring(matrix(1:32,8)))
x = st_sfc(st_linestring(matrix(1:32,8)), st_linestring(matrix(1:8,2)))
st_zm(x)
a = st_sf(a = 1:2, geom=x)
st_zm(a)
```

Description

Return 'z' range of a simple feature or simple feature set

Usage

```r
## S3 method for class 'z_range'
is.na(x)

st_z_range(obj, ...)

## S3 method for class 'POINT'
st_z_range(obj, ...)

## S3 method for class 'MULTIPOINT'
st_z_range(obj, ...)

## S3 method for class 'LINESTRING'
st_z_range(obj, ...)

## S3 method for class 'POLYGON'
st_z_range(obj, ...)

## S3 method for class 'MULTILINESTRING'
```
st_z_range(obj, ...)

## S3 method for class 'MULTIPOLYGON'
st_z_range(obj, ...)

## S3 method for class 'GEOMETRYCOLLECTION'
st_z_range(obj, ...)

## S3 method for class 'MULTISURFACE'
st_z_range(obj, ...)

## S3 method for class 'MULTICURVE'
st_z_range(obj, ...)

## S3 method for class 'CURVEPOLYGON'
st_z_range(obj, ...)

## S3 method for class 'COMPOUNDCURVE'
st_z_range(obj, ...)

## S3 method for class 'POLYHEDRALSURFACE'
st_z_range(obj, ...)

## S3 method for class 'TIN'
st_z_range(obj, ...)

## S3 method for class 'TRIANGLE'
st_z_range(obj, ...)

## S3 method for class 'CIRCULARSTRING'
st_z_range(obj, ...)

## S3 method for class 'sfc'
st_z_range(obj, ...)

## S3 method for class 'sf'
st_z_range(obj, ...)

## S3 method for class 'numeric'
st_z_range(obj, ..., crs = NA_crs_)

NA_z_range_

Arguments

x        object of class z_range
obj      object to compute the z range from
...      ignored
**summary.sfc**

```
crs object of class crs, or argument to st_crs, specifying the CRS of this bounding box.

Format
An object of class z_range of length 2.

Details
NA_z_range_ represents the missing value for a z_range object

Value
a numeric vector of length two, with zmin and zmax values; if obj is of class sf or sfc the object returned has a class z_range

Examples
```
a = st_sf(a = 1:2, geom = st_sfc(st_point(0:2), st_point(1:3)), crs = 4326)
st_z_range(a)
st_z_range(c(zmin = 16.1, zmax = 16.6), crs = st_crs(4326))
```

**summary.sfc**  
*Summarize simple feature column*

**Description**
Summarize simple feature column

**Usage**
```
## S3 method for class 'sfc'
summary(object, ..., maxsum = 7L, maxp4s = 10L)
```

**Arguments**
```
object object of class sfc
... ignored
maxsum maximum number of classes to summarize the simple feature column to
maxp4s maximum number of characters to print from the PROJ string
```
### tibble

**Summarize simple feature type for tibble**

**Description**

Summarize simple feature type for tibble

Summarize simple feature item for tibble

**Usage**

```r
type_sum.sfc(x, ...)

obj_sum.sfc(x)

pillar_shaft.sfc(x, ...)
```

**Arguments**

- `x`: object of class sfc
- `...`: ignored

**Details**

see `type_sum`

### tidyverse

**Tidyverse methods for sf objects (remove .sf suffix!)**

**Description**

Tidyverse methods for sf objects. Geometries are sticky, use `as.data.frame` to let dplyr’s own methods drop them. Use these methods without the .sf suffix and after loading the tidyverse package with the generic (or after loading package tidyverse).

**Usage**

```r
dplyr_reconstruct.sf(data, template)

filter.sf(.data, ..., .dots)

arrange.sf(.data, ..., .dots)

group_by.sf(.data, ..., add = FALSE)

ungroup.sf(x, ...)
```
tidyverse

rowwise.sf(x, ...)

mutate.sf(.data, ..., .dots)

transmute.sf(.data, ..., .dots)

select.sf(.data, ...)

rename.sf(.data, ...)

slice.sf(.data, ..., .dots)

summarise.sf(.data, ..., .dots, do_union = TRUE, is_coverage = FALSE)

distinct.sf(.data, ..., .keep_all = FALSE)

gather.sf(
  data,
  key,
  value,
  ...,
  na.rm = FALSE,
  convert = FALSE,
  factor_key = FALSE
)

pivot_longer.sf(
  data,
  cols,
  names_to = "name",
  names_prefix = NULL,
  names_sep = NULL,
  names_pattern = NULL,
  names_ptypes = NULL,
  names_transform = NULL,
  names_repair = "check_unique",
  values_to = "value",
  values_drop_na = FALSE,
  values_ptypes = NULL,
  values_transform = NULL,
  ...
)

pivot_wider.sf(
  data,
  id_cols = NULL,
  names_from,
names_prefix = "",
names_sep = "_",
names_glue = NULL,
names_sort = FALSE,
names_repair = "check_unique",
values_from,
values_fill = NULL,
values_fn = NULL,
...
)

spread.sf(
data,
key,
value,
fill = NA,
convert = FALSE,
drop = TRUE,
sep = NULL
)

sample_n.sf(tbl, size, replace = FALSE, weight = NULL, .env = parent.frame())

sample_frac.sf(
  tbl,
  size = 1,
  replace = FALSE,
  weight = NULL,
  .env = parent.frame()
)

nest.sf(.data, ...)

separate.sf(
data,
col,
into,
sep = "[^[:alnum:]]+",
remove = TRUE,
convert = FALSE,
extra = "warn",
fill = "warn",
...
)

separate_rows.sf(data, ..., sep = "[^[:alnum:]]+", convert = FALSE)

unite.sf(data, col, ..., sep = "_", remove = TRUE)
unnest.sf(data, ..., .preserve = NULL)

drop_na.sf(x, ...)

inner_join.sf(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ...)

left_join.sf(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ...)

right_join.sf(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ...)

full_join.sf(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ...)

semi_join.sf(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ...)

anti_join.sf(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ...)

Arguments

data see original function docs

template see original function docs

.data data object of class sf

... other arguments

.dots see corresponding function in package dplyr

add see corresponding function in dplyr

x, y A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See Methods, below, for more details.

do_union logical; in case summary does not create a geometry column, should geometries be created by unioning using st_union, or simply by combining using st_combine? Using st_union resolves internal boundaries, but in case of unioning points, this will likely change the order of the points; see Details.

is_coverage logical; if do_union is TRUE, use an optimized algorithm for features that form a polygonal coverage (have no overlaps)

.keep_all see corresponding function in dplyr

key see original function docs

value see original function docs

na.rm see original function docs

convert see separate_rows

factor_key see original function docs

cols see original function docs

names_to see original function docs

names_prefix see original function docs

names_sep see original function docs
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>by</code></td>
<td>A join specification created with <code>join_by()</code>, or a character vector of variables to join by. If NULL, the default, <code>*_join()</code> will perform a natural join, using all variables in common across x and y. A message lists the variables so that you can check they’re correct; suppress the message by supplying <code>by</code> explicitly. To join on different variables between x and y, use a <code>join_by()</code> specification. For example, <code>join_by(a == b)</code> will match x$a to y$b. To join by multiple variables, use a <code>join_by()</code> specification with multiple expressions. For example, <code>join_by(a == b, c == d)</code> will match x$a to y$b and</td>
</tr>
<tr>
<td><code>names_pattern</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>names_ptypes</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>names_transform</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>names_repair</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>values_to</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>values_drop_na</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>values_ptypes</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>values_transform</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>id_cols</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>names_from</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>names_glue</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>names_sort</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>values_from</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>values_fill</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>values_fn</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>fill</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>drop</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>sep</code></td>
<td>see <code>separate_rows</code></td>
</tr>
<tr>
<td><code>tbl</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>size</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>replace</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>weight</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>.env</code></td>
<td>see original function docs</td>
</tr>
<tr>
<td><code>col</code></td>
<td>see <code>separate</code></td>
</tr>
<tr>
<td><code>into</code></td>
<td>see <code>separate</code></td>
</tr>
<tr>
<td><code>remove</code></td>
<td>see <code>separate</code></td>
</tr>
<tr>
<td><code>extra</code></td>
<td>see <code>separate</code></td>
</tr>
<tr>
<td><code>.preserve</code></td>
<td>see <code>unnest</code></td>
</tr>
</tbody>
</table>
x$c to y$d. If the column names are the same between x and y, you can shorten this by listing only the variable names, like join_by(a, c).

join_by() can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins.

For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, by = c("a", "b") joins x$a to y$a and x$b to y$b. If variable names differ between x and y, use a named character vector like by = c("x_a" = "y_a", "x_b" = "y_b").

To perform a cross-join, generating all combinations of x and y, see cross_join().

copy

If x and y are not from the same data source, and copy is TRUE, then y will be copied into the same src as x. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.

suffix

If there are non-joined duplicate variables in x and y, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.

Details

select keeps the geometry regardless whether it is selected or not; to deselect it, first pipe through as.data.frame to let dplyr's own select drop it.

In case one or more of the arguments (expressions) in the summarise call creates a geometry list-column, the first of these will be the (active) geometry of the returned object. If this is not the case, a geometry column is created, depending on the value of do_union.

In case do_union is FALSE, summarise will simply combine geometries using c.sfg. When polygons sharing a boundary are combined, this leads to geometries that are invalid; see for instance https://github.com/r-spatial/sf/issues/681.

distinct gives distinct records for which all attributes and geometries are distinct; st_equals is used to find out which geometries are distinct.

nest assumes that a simple feature geometry list-column was among the columns that were nested.

Value

an object of class sf

Examples

if (require(dplyr, quietly = TRUE)) {
  nc = read_sf(system.file("shape/nc.shp", package="sf"))
  nc %>% filter(AREA > .1) %>% plot()
  # plot 10 smallest counties in grey:
  st_geometry(nc) %>% plot()
  nc %>% select(AREA) %>% arrange(AREA) %>% slice(1:10) %>% plot(add = TRUE, col = "grey")
  title("the ten counties with smallest area")
  nc2 <- nc %>% mutate(area10 = AREA/10)
  nc %>% slice(1:2)
}
# plot 10 smallest counties in grey:
if (require(dplyr, quietly = TRUE)) {
  st_geometry(nc) %>% plot()
tidyverse

```r
nc %>% select(AREA) %>% arrange(AREA) %>% slice(1:10) %>% plot(add = TRUE, col = 'grey')
title("the ten counties with smallest area")
if (require(dplyr, quietly = TRUE)) {
  nc$area_cl = cut(nc$AREA, c(0, .1, .12, .15, .25))
  nc %>% group_by(area_cl) %>% class()
}
if (require(dplyr, quietly = TRUE)) {
  nc2 <- nc %>% mutate(area0 = AREA/10)
}
if (require(dplyr, quietly = TRUE)) {
  nc2 <- nc %>% transmute(AREA = AREA/10, geometry = geometry) %>% class()
  nc %>% transmute(AREA = AREA/10) %>% class()
}
if (require(dplyr, quietly = TRUE)) {
  nc %>% select(SID74, SID79) %>% names()
  nc %>% select(SID74, SID79, geometry) %>% names()
  nc %>% select(SID74, SID79) %>% class()
  nc %>% select(SID74, SID79, geometry) %>% class()
}
if (require(dplyr, quietly = TRUE)) {
  nc %>% slice(1:2)
}
if (require(dplyr, quietly = TRUE)) {
  nc$area_cl = cut(nc$AREA, c(0, .1, .12, .15, .25))
  nc.g <- nc %>% group_by(area_cl)
  nc.g %>% summarise(mean(AREA))
  nc.g %>% summarise(mean(AREA)) %>% plot(col = grey(3:6 / 7))
  nc %>% as.data.frame %>% summarise(mean(AREA))
}
if (require(dplyr, quietly = TRUE)) {
  nc[c(1:100, 1:10), ] %>% distinct() %>% nrow()
}
if (require(dplyr, quietly = TRUE) && require(dplyr, quietly = TRUE)) {
  nc %>% select(SID74, SID79) %>% gather("VAR", "SID", -geometry) %>% summary()
}
if (require(dplyr, quietly = TRUE) && require(dplyr, quietly = TRUE)) {
  nc$row = 1:100 # needed for spread to work
  nc %>% select(SID74, SID79, geometry, row) %>%
  gather("VAR", "SID", -geometry, -row) %>%
  spread(VAR, SID) %>% head()
}
if (require(dplyr, quietly = TRUE) && require(dplyr, quietly = TRUE)) {
  storms.sf = st_as_sf(storms, coords = c("long", "lat"), crs = 4326)
  x <- storms.sf %>% group_by(name, year) %>% nest
  trs = lapply(x$data, function(tr) st_cast(st_combine(tr), "LINESTRING")[[1]]) %>%
    st_sf(crs = 4326)
  trs.sf = st_sf(x[,1:2], trs)
  plot(trs.sf["year"], axes = TRUE)
```
transform.sf

**transform method for sf objects**

**Description**

Can be used to create or modify attribute variables; for transforming geometries see `st_transform`, and all other functions starting with `st_`.

**Usage**

```
## S3 method for class 'sf'
transform(\_data, ...)
```

**Arguments**

- `\_data` object of class sf
- `...` Further arguments of the form new\_variable=expression

**Examples**

```r
a = data.frame(x1 = 1:3, x2 = 5:7)
st\_geometry(a) = st\_sfc(st\_point(c(0,0)), st\_point(c(1,1)), st\_point(c(2,2)))
transform(a, x1\_sq = x1^2)
transform(a, x1\_x2 = x1*x2)
```

---

**valid**

**Check validity or make an invalid geometry valid**

**Description**

Checks whether a geometry is valid, or makes an invalid geometry valid

**Usage**

```
st\_is\_valid(x, ...)
```

```
## S3 method for class 'sfc'
st\_is\_valid(x, ..., NA_on_exception = TRUE, reason = FALSE)
```

```
## S3 method for class 'sf'
st\_is\_valid(x, ...)
```

```
## S3 method for class 'sfg'
st\_is\_valid(x, ...)
```

```
st\_make\_valid(x, ...)
```
## S3 method for class 'sfg'

\texttt{st_make_valid(x, ...)}

## S3 method for class 'sfc'

\texttt{st_make_valid(}
\begin{itemize}
  \item \texttt{x},
  \item \texttt{...},
\end{itemize}
\begin{itemize}
  \item \texttt{oriented = FALSE,}
  \item \texttt{s2_options = s2::s2_options(snap = s2::s2_snap_precision(1e+07), ...),}
  \item \texttt{geos_method = "valid_structure",}
  \item \texttt{geos_keep_collapsed = TRUE}
\end{itemize}
\texttt{)}

### Arguments

- \texttt{x} object of class sfg, sf or sf
- \texttt{...} passed on to \texttt{s2_options}
- \texttt{NA_on_exception} logical; if TRUE, for polygons that would otherwise raise a GEOS error (exception, e.g. for a POLYGON having more than zero but less than 4 points, or a LINESTRING having one point) return an NA rather than raising an error, and suppress warning messages (e.g. about self-intersection); if FALSE, regular GEOS errors and warnings will be emitted.
- \texttt{reason} logical; if TRUE, return a character with, for each geometry, the reason for invalidity, NA on exception, or "Valid Geometry" otherwise.
- \texttt{oriented} logical; only relevant if \texttt{st_is_longlat(x)} is TRUE; see \texttt{s2}
- \texttt{s2_options} only relevant if \texttt{st_is_longlat(x)} is TRUE; options for \texttt{s2_rebuild}, see \texttt{s2_options} and Details.
- \texttt{geos_method} character; either "valid_linework" (Original method, combines all rings into a set of noded lines and then extracts valid polygons from that linework) or "valid_structure" (Structured method, first makes all rings valid then merges shells and subtracts holes from shells to generate valid result. Assumes that holes and shells are correctly categorized.) (requires GEOS >= 3.10.1)
- \texttt{geos_keep_collapsed} logical; When this parameter is not set to FALSE, the "valid_structure" method will keep any component that has collapsed into a lower dimensionality. For example, a ring collapsing to a line, or a line collapsing to a point (requires GEOS >= 3.10.1)

### Details

For projected geometries, \texttt{st_make_valid} uses the \texttt{lwgeom_makevalid} method also used by the PostGIS command \texttt{ST_makevalid} if the GEOS version linked to is smaller than 3.8.0, and otherwise the version shipped in GEOS; for geometries having ellipsoidal coordinates \texttt{s2::s2_rebuild} is being used.

If \texttt{s2_options} is not specified and \texttt{x} has a non-zero precision set, then this precision value will be used as the value in \texttt{s2_snap_precision}, passed on to \texttt{s2_options}, rather than the 1e7 default.
Value

st_is_valid returns a logical vector indicating for each geometries of x whether it is valid. st_make_valid returns an object with a topologically valid geometry.

Object of the same class as x

Examples

p1 = st_as_sfc("POLYGON((0 0, 0 10, 10 0, 10 10, 0 0))")
st_is_valid(p1)
st_is_valid(st_sfc(st_point(0:1), p1[[1]]), reason = TRUE)
library(sf)
x = st_sfc(st_polygon(list(rbind(c(0,0),c(0.5,0),c(0.5,0.5),c(0.5,0),c(1,0),c(1,1),c(0,1),c(0,0)))))
suppressWarnings(st_is_valid(x))
y = st_make_valid(x)
st_is_valid(y)
y %>% st_cast()

Description

vctrs methods for sf objects

Usage

vec_ptype2.sfc(x, y, ...)

## Default S3 method:
vec_ptype2.sfc(x, y, ..., x_arg = "x", y_arg = "y")

## S3 method for class 'sfc'
vec_ptype2.sfc(x, y, ...)

vec_cast.sfc(x, to, ...)

## S3 method for class 'sfc'
vec_cast.sfc(x, to, ...)

## Default S3 method:
vec_cast.sfc(x, to, ...)

Arguments

x, y Vector types.

... These dots are for future extensions and must be empty.

x_arg, y_arg Argument names for x and y.

to Type to cast to. If NULL, x will be returned as is.
Index

* datasets
  db_drivers, 10
  extension_map, 10
  prefix_map, 43
  st_agr, 57
  st_bbox, 66
  st_crs, 76
  st_m_range, 91
  st_z_range, 113

* data
  nc, 35
  .degAxis (stars), 55
  .get_layout (stars), 55
  .image_scale (stars), 55
  .image_scale_factor (stars), 55
  .stop_geos (internal), 32
  ?join_by, 121
  [.data.frame, 47, 48
  [.sf (sf), 46
  $.crs (st_crs), 76
  aggregate, 4, 5, 70, 97
  aggregate (aggregate.sf), 4
  aggregate.sf, 4
  anti_join.sf (tidyverse), 116
  arrange.sf (tidyverse), 116
  as, 6
  as.data.frame, 100, 116
  as.matrix.sfg (st), 53
  as.matrix.sgbp (sgbp), 52
  as_Spatial (as), 6
  as_Spatial(), 6
  bind, 7
  bind_cols, 7
  bpy.colors, 42
  c, 23
  c.sfg, 5, 23, 121
  c.sfg (st), 53
  cbind, 7
  cbind.sf (bind), 7
  chull, 29
  classIntervals, 40
  coerce,crs,CRS-method (as), 6
  coerce,sf,Spatial-method (as), 6
  coerce,sfc,Spatial-method (as), 6
  coerce,Spatial,sf-method (as), 6
  coerce,Spatial,sfc-method (as), 6
  cross_join(), 121
  data.frame, 7
  db_drivers, 10
  dbDataType,DBIOBJECT,sf-method
    (dbDataType,PostgreSQLConnection,sf-method), 8
  dbDataType,PostgreSQLConnection,sf-method, 8
  dbWriteTable, 111, 112
  dbWriteTable,DBIOBJECT,character,sf-method
    (dbWriteTable,PostgreSQLConnection,character,sf-method), 8
  dbWriteTable,PostgreSQLConnection,character,sf-method, 8
  dim.sgbp (sgbp), 52
  distinct.sf, 21
  distinct.sf (tidyverse), 116
  dotsMethods, 7
  dplyr_reconstruct.sf (tidyverse), 116
  drop_na.sf (tidyverse), 116
  extension_map, 10
  filter.sf (tidyverse), 116
  format, 67
  format.bbox (st_bbox), 66
  format.crs (st_crs), 76
  format.sfg (st), 53
  full_join.sf (tidyverse), 116
  gather.sf (tidyverse), 116

126
gdal, 10
gdal_addo, 14, 16
gdal_create (gdal), 10
gdal_crs (gdal), 10
gdal_extract (gdal), 10
gdal_inv_geotransform (gdal), 10
gdal_metadata (gdal), 10
gdal_polygonize (gdal), 10
gdal_rasterize (gdal), 10
gdal_read (gdal), 10
gdal_read_mdim (gdal), 10
gdal_subdatasets (gdal), 10
gdal_utils, 15, 15
gdal_write (gdal), 10
gdal_write_mdim (gdal), 10
geos_binary_ops, 17
geos_binary_pred, 19
geos_combine, 22
geos_measures, 23
geos_query, 25
geos Unary, 26
get_key_pos (plot), 37
getwd, 101

head.sfg (st), 53

Id, 99
inner_join.sf (tidyverse), 116
internal, 32
interpolate_aw, 32
intersect, 19
is.na.bbox (st_bbox), 66
is.na.crs (st_crs), 76
is.na.m_range (st_m_range), 91
is.na.z_range (st_z_range), 113
is_driver_available, 33
is_driver_can, 34
is_geometry_column, 34

join_by(), 120, 121
left_join, 86
left_join.sf (tidyverse), 116

map, 61
merge, 86
merge.sf, 35
mutate.sf (tidyverse), 116

NA_agr_ (st_agr), 57
NA_bbox_ (st_bbox), 66
NA_crs_ (st_crs), 76
NA_m_range_ (st_m_range), 91
NA_z_range_ (st_z_range), 113
nc, 35
nest.sf (tidyverse), 116

obj_sum.sfc (tibble), 116
Ops, 36

par, 41, 42
pillar_shaft.sfc (tibble), 116
pivot_longer.sf (tidyverse), 116
pivot_wider.sf (tidyverse), 116
plot, 37, 40
plot_window, 41
plot_sf, 40
plot_sf (plot), 37
polypath, 41
prefix_map, 43
print.sf (sf), 46
print.sfg (st), 53
print.sgbp (sgbp), 52
proj_tools, 43

rainbow, 40
rawToHex, 45
rbind, 7
rbind.sf (bind), 7
read_sf (st_read), 98
recenter, 105
rename.sf (tidyverse), 116
right_join.sf (tidyverse), 116
rowwise.sf (tidyverse), 116
RPostgres::Postgres(), 111

s2, 45, 124
s2_distance, 24
s2_distance_matrix, 24
s2_options, 17, 18, 20, 124
s2_rebuild, 46, 124
sample, 103
sample_frac.sf (tidyverse), 116
sample.n.sf (tidyverse), 116
select.sf (tidyverse), 116
semi_join.sf (tidyverse), 116
separate, 120
separate.sf (tidyverse), 116
separate_rows, 119, 120
separate_rows.sf (tidyverse), 116
set_units, 25
setdiff, 19
sf, 4, 46, 77, 81, 84, 90, 100, 119, 121
sf-defunct, 48
sf.colors (plot), 37
sf_add_proj_units (sf_project), 51
sf_extSoftVersion, 50
sf_proj_info (st_transform), 106
sf_proj_network (proj_tools), 43
sf_proj_pipelines (proj_tools), 43
sf_proj_search_paths (proj_tools), 43
sf_project, 51, 108
sf_use_s2 (s2), 45
sfc, 49, 61, 77, 81, 84, 90, 95
sfc_GEOMETRYCOLLECTION (sfc), 49
sfc_LINESTRING (sfc), 49
sfc_MULTILINESTRING (sfc), 49
sfc_MULTIPOINT (sfc), 49
sfc_MULTIPOINT (sf), 49
sfc_MULTIPOLYGON (sf), 49
sfc_POINT (sfc), 49
sfc_POLYGON (sf), 49
sgbp, 21, 52
slice.sf (tidyverse), 116
spread.sf (tidyverse), 116
st, 53
st_agr, 57
st_agr <- (st_agr), 57
st_area (geos_measures), 23
st_as_binary, 47, 50, 58, 64, 97
st_as_grob, 59
st_as_s2 (s2), 45
st_as_sf, 49, 59, 99
st_as_sfc, 47, 48, 62, 68
st_as_text, 65
st_axis_order (st_crs), 76
st_bbox, 66, 75
st_bind_cols (bind), 7
st_boundary (geos_unary), 26
st_break_antimeridian, 68, 108
st_buffer (geos_unary), 26
st_can_transform (st_transform), 106
st_cast, 25, 69, 73
st_cast(), 6
st_cast_sfc_default, 72
st_centroid (geos_unary), 26
st_collection_extract, 72
st_combine, 119
st_combine (geos_combine), 22
st_concave_hull (geos_unary), 26
st_contains, 86
st_contains (geos_binary_pred), 19
st_contains_properly, 86
st_contains_properly (geos_binary_pred), 19
st_convex_hull (geos_unary), 26
st_coordinates, 74
st_covered_by, 86
st_covered_by (geos_binary_pred), 19
st_covers, 86
st_covers (geos_binary_pred), 19
st_crop, 75
st_crosses, 86
st_crs, 14, 47, 67, 76, 84, 92, 107, 115
st_crs(), 64
st_crs <- (st_crs), 76
st_delete (st_write), 110
st_difference, 23
st_difference (geos_binary_ops), 17
st_dimension, 25
st_dimension (geos_query), 25
st_disjoint, 86
st_disjoint (geos_binary_pred), 19
st_distance (geos_measures), 23
st_drivers, 33, 34, 79, 101, 112
st_drop_geometry (st_geometry), 79
st_equals, 87, 121
st_equals (geos_binary_pred), 19
st_equals_exact, 86
st_equals_exact (geos_binary_pred), 19
st_filter (st_join), 85
st_geod_area, 25
st_geod_distance, 24
st_geod_segmentize, 28
st_geometry, 79
st_geometry <- (st_geometry), 79
st_geometry_type, 81
st_geometries (st), 53
st_graticule, 42, 82
st_inscribed_circle (geos_unary), 26
st_interpolate_aw (interpolate_aw), 32
st_intersection, 21, 23, 102
st_intersection (geos_binary_ops), 17
st_intersects, 18, 59, 86
INDEX

st_intersects (geos_binary_pred), 19
st_is, 83
st_is_empty (geos_query), 25
st_is_longlat, 84
st_is_simple (geos_query), 25
st_is_valid (valid), 123
st_is_within_distance, 87
st_is_within_distance (geos_binary_pred), 19
st_jitter, 85
st_join, 5, 85
st_layers, 16, 88, 100, 101
st_length (geos_measures), 23
st_line_merge (geos_unary), 26
st_line_sample, 89
st_linestring (st), 53
st_m_range, 91
st_make_grid, 90
st_make_valid (valid), 123
st_minimum_rotated_rectangle (geos_unary), 26
st_multilinestring (st), 53
st_multipoint (st), 53
st_multipolygon (st), 53
st_nearest_feature, 87, 93, 95
st_nearest_points, 94, 94
st_node (geos_unary), 26
st_normalize, 96
st_overlaps, 87
st_overlaps (geos_binary_pred), 19
st_point, 61
st_point (st), 53
st_point_on_surface (geos_unary), 26
st_polygon (st), 53
st_polygonize (geos_unary), 26
st_precision, 97
st_precision<- (st_precision), 97
st_read, 47, 49, 50, 98
st_read_db (sf-defunct), 48
st_read_db (sf-defunct), 48
st_relate, 21, 86, 102
st_reverse (geos_unary), 26
st_sample, 103
st_segmentize (geos_unary), 26
st_set_agr (st_agr), 57
st_set_crs (st_crs), 76
st_set_geometry (st_geometry), 79
st_set_precision, 36
st_set_precision (st_precision), 97
st_sf, 7, 60
st_sf (sf), 46
st_sfc (sfc), 49
st_shift_longitude, 105
st_simplify (geos_unary), 26
st_snap (geos_binary_ops), 17
st_sym_difference, 23
st_sym_difference (geos_binary_ops), 17
stTouches, 87
stTouches (geos_binary_pred), 19
st_transform, 106, 123
st_transform_proj, 108
st_triangulate (geos_unary), 26
st_triangulate_constrained (geos_unary), 26
st_union, 5, 19, 119
st_union (geos_combine), 22
st_viewport, 109
st_voronoi (geos_unary), 26
st_within, 87
st_within (geos_binary_pred), 19
st_wrap_dateline (st_transform), 106
st_write, 49, 97, 110
st_write_db (sf-defunct), 48
st_z_range, 113
st_zm, 112
st_zm(), 6
stars, 55
summarise, 70, 97
summarise (tidyverse), 116
summary.sfc, 115
t.sgbp (sgbp), 52
tibble, 116
tidyverse, 116
transform.sf, 123
transmute.sf (tidyverse), 116
type_sum, 116
type_sum.sfc (tibble), 116
ungroup.sf (tidyverse), 116
unite.sf (tidyverse), 116
units, 78
unnest, 120
unnest.sf (tidyverse), 116
valid, 123
vctrs, 125
vec_cast.sfc(vctrs), 125
vec_pType2.sfc(vctrs), 125
viewport, 109

write_sf, 97
write_sf(st_write), 110