Package ‘sf’

July 5, 2019

Version 0.7-6
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.
License GPL-2 | MIT + file LICENSE
BugReports https://github.com/r-spatial/sf/issues/
Depends methods, R (>= 3.3.0)
Imports classInt (>= 0.2-1), DBI (>= 0.8), graphics, grDevices, grid, magrittr, Rcpp (>= 0.12.18), stats, tools, units (>= 0.6-0), utils
Suggests blob, covr, dplyr (>= 0.8-0), ggplot2, knitr, lwgeom (>= 0.1-5), maps, maptools, mapview, microbenchmark, odbc, pillar, pool, raster, rgdal, rgeos, rlang, rmarkdown, RPostgres (>= 1.1.0), RPostgreSQL, RSQlite, sp (>= 1.2-4), spatstat, stars (>= 0.2-0), testthat, tibble (>= 1.4.1), tidyR (>= 0.7-2), tidyselect, tm (>= 2.0)
LinkingTo Rcpp
VignetteBuilder knitr
Encoding UTF-8
RoxygenNote 6.1.1
SystemRequirements C++11, GDAL (>= 2.0.1), GEOS (>= 3.4.0), PROJ (>= 4.8.0)
R topics documented:

'sets.R' 'crop.R' 'gdal_utils.R' 'nearest.R' 'normalize.R'
'deprecated.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]

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

Repository: CRAN

Date/Publication: 2019-07-05 12:30:08 UTC

R topics documented:

aggregate.sf ............................................................... 4
as ................................................................. 5
bind ................................................................. 6
dbDataType,PostgreSQLConnection,sf-method ........................................ 7
db_drivers ......................................................... 8
extension_map ...................................................... 8
gdal ................................................................. 9
gdal_utils ........................................................ 11
geos_binary_ops .................................................. 12
geos_binary_pred .................................................. 14
geos_combine ....................................................... 16
geos_measures ..................................................... 17
geos_query .......................................................... 19
geos_unary ............................................................. 20
internal ............................................................ 24
is_driver_available .................................................. 24
is_driver_can ....................................................... 24
is_geometry_column ........................................... 25
merge.sf ............................................................. 25
nc ................................................................. 26
Ops ................................................................. 26
plot ................................................................. 27
prefix_map .......................................................... 31
rawToHex ............................................................ 31
sf ................................................................. 32
sf-deprecated ...................................................... 34
R topics documented:

sfc ................................................................. 34
sf_extSoftVersion .................................................. 35
sf_project ............................................................. 36
sgbp ................................................................. 36
st ................................................................. 37
stars ............................................................... 40
st_agr .............................................................. 41
st_as_binary ....................................................... 41
st_as_grob .......................................................... 43
st_as_sf ............................................................. 43
st_as_sfc ........................................................... 45
st_as_text .......................................................... 47
st_bbox ............................................................. 48
st_cast .............................................................. 51
st_cast_sfc_default ............................................... 53
st_collection_extract .......................................... 54
st_coordinates .................................................... 55
st_crop ............................................................. 56
st_crs .............................................................. 57
st_drivers .......................................................... 59
st_geometry ......................................................... 60
st_geometry_type .................................................. 61
st_graticule ....................................................... 62
st_interpolate_aw ............................................... 63
st_is ............................................................... 64
st_is_longlat ...................................................... 65
st_jitter ........................................................... 65
st_join ............................................................. 66
st_layers ........................................................... 67
st_line_sample .................................................... 68
st_make_grid ....................................................... 69
st_nearest_feature .............................................. 70
st_nearest_points ................................................. 71
st_normalize ....................................................... 72
st_precision ......................................................... 73
st_read ............................................................. 74
st_relate ........................................................... 77
st_sample .......................................................... 78
st_transform ....................................................... 80
st_viewport ......................................................... 82
st_write ........................................................... 83
st_zm ............................................................... 86
summary.sfc ...................................................... 86
tibble ............................................................. 87
tidyverse .......................................................... 87

Index 92
aggregate.sf

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
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.1 for ids[[i]]; see aggregate.

Note

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

```r
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)))
```

---

as

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", 1:length(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

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

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

Examples

nc <- st_read(system.file("shape/nc.shp", package="sf"))
# 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

## 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

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

- **dbObj**: DBIOObject driver or connection.
- **obj**: Object to convert

---

**db_drivers**  
*Drivers for which update should be TRUE by default*

---

Description

Drivers for which update should be TRUE by default

Usage

**db_drivers**

Format

An object of class character of length 12.

---

**extension_map**  
*Map extension to driver*

---

Description

Map extension to driver

Usage

**extension_map**

Format

An object of class list of length 24.
gdal functions to interact with gdal not meant to be called directly by users (but e.g. by stars::st_stars)

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

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())

Arguments

x character vector, possibly of length larger than 1 when more than one raster is read

... ignored

options character; raster layer read options

driver character; 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.
- file character; file name
- type gdal write type
- geotransform length 6 numeric vector with GDAL geotransform parameters.
- update logical; TRUE if in an existing raster file pixel values shall be updated.
- gt double vector of length 6
- domain_item character vector of length 0, 1 (with domain), or 2 (with domain and item); use "" for the default domain, use NA_character_ to query the domain names.
- parse logical; should metadata be parsed into a named list (TRUE) or returned as character data?
- name logical; retrieve name of subdataset? If FALSE, retrieve description
- mask stars object with NA mask (0 where NA), or NULL
- use_integer 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.
- breaks numeric vector with break values for contour polygons (or lines)
- use_contours logical;
- contour_lines logical;
- connect8 logical; if TRUE use 8 connection algorithm, rather than 4
- sf object of class sf

Details

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_)
```
**Description**

Native interface to gdal utils

**Usage**

```r
gdal_utils(util = "info", source, destination, options = character(0),
quiet = FALSE, processing = character(0),
colorfilename = character(0))
```

**Arguments**

- `util` character; one of `info`, `warp`, `rasterize`, `translate`, `vectortranslate`, `buildvrt`, `demprocessing`, `nearblack`, `grid`
- `source` character; name of input layer(s); for `warp` or `buildvrt` this can be more than one
- `destination` character; name of output layer
- `options` character; raster layer read options
- `quiet` logical; if TRUE, suppress printing of output for `info`
- `processing` character; processing options for `demprocessing`
- `colorfilename` character; name of color file for `demprocessing` (mandatory if `processing="color-relief"`)

**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.
Geometric operations on pairs of simple feature geometry sets

Description

Perform geometric set operations with simple feature geometry collections

Usage

```r
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`
- `tolerance`: 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 \texttt{origins} with indexes of all overlapping features.

When \texttt{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 \texttt{st\_difference\_sfc} method with a single argument returns an object with an "\texttt{idx}" attribute with the orginal index for returned geometries.

**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 \texttt{sf}, the matching attributes of the original object(s) are added. The \texttt{sfc} geometry list-column returned carries an attribute \texttt{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 \texttt{st\_intersects} instead of \texttt{st\_intersection}.

**See Also**

\texttt{st\_union} for the union of simple features collections; \texttt{intersect} and \texttt{setdiff} for the base R set operations.

**Examples**

```r
set.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")
summary(lengths(st_overlaps(s, s))) # includes self-counts!
summary(lengths(st_overlaps(d, d)))
summary(lengths(st_overlaps(i, i)))
sf = st_sf(s)
i = st_intersection(sf) # all intersections
```
**Description**

Geometric binary predicates on pairs of simple feature geometry sets

**Usage**

- `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)`
- `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)`
- `st_covers(x, y, sparse = TRUE, prepared = TRUE)`
- `st_covered_by(x, y, sparse = TRUE, prepared = TRUE)`
- `st_equals_exact(x, y, par, sparse = TRUE, prepared = FALSE)`
- `st_is_within_distance(x, y, dist, sparse = TRUE)`

**Arguments**

- `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.
- `...`: ignored
preparation logical; prepare geometry for x, before looping over y? See Details.
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 http://r-spatial.org/r/2017/06/22/spatial-index.html. Specifically, st_intersects, st_disjoint, st_touches st_crosses, st_within, st_contains, st_contains_properly, st_overlaps, st_equals, st_covers and st_covered_by all build spatial indexes for more efficient geometry calculations. st_relate, st_equals_exact, and st_is_within_distance do not.

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.

'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 integer(0) 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

pts = 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(pts, pol))
(mat = st_intersects(pts, 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, pts)[[1]]

---

**geos_combine**

**Combine or union feature geometries**

**Description**

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

**Usage**

st_combine(x)

st_union(x, y, ..., by_feature = 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

**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 gUnaryUnion, the latter to 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

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

geos_measures  Compute geometric measurements

Description

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

Usage

st_area(x, ...)
st_length(x)

st_distance(x, y, ..., dist_fun, by_element = FALSE,
            which = "Euclidean", par = 0, tolerance = 0)

Arguments

x          object of class sf, sfc or sfg
...         ignored
y          object of class sf, sfc or sfg, defaults to x
dist_fun    deprecated
by_element  logical; if TRUE, return a vector with distance between the first elements of x and y, the second, etc. if FALSE, return the dense matrix with all pairwise distances.
which       character; for Cartesian coordinates only: one of Euclidian, Haussdorf or Frechet; for geodetic coordinates, great circle distances are computed; see details
par for which equal to Haussdorff or Frechet, optionally use a value between 0 and 1 to densify the geometry.

tolerance 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


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, MULTIPPOINT, 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 of length x or y, the shorter one being recycled. Distances involving empty geometries are NA.

See Also

st_dimension, st_cast to convert geometry types.

Examples

```r
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 = CRS(4326))
st_length(line)

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

poly = st_polygon(list(outer, hole1, hole2))
mpoly = st_multipolygon(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)))
```
Description

Dimension, simplicity, validity or is_empty queries on simple feature geometries

Usage

st_dimension(x, NA_if_empty = TRUE)

st_is_simple(x)

st_is_empty(x)

st_is_valid(x, NA_on_exception = TRUE, reason = FALSE)

Arguments

x object of class sf, sfc or sfg

NA_if_empty logical; if TRUE, return NA for empty geometries

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.

reason logical; if TRUE, return a character with, for each geometry, the reason for invalidity, NA on exception, or "Valid Geometry" otherwise.

Value

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.

st_is_simple returns a logical vector, indicating for each geometry whether it is simple (e.g., not self-intersecting)

st_is_empty returns for each geometry whether it is empty

st_is_valid returns a logical vector indicating for each geometries of x whether it is valid.
Examples

```r
x = st_sfc(
    st_point(c(0:1)),
    st_linestring(rbind(c(0,0),c(1,1))),
    st_polygon(list(rbind(c(0,0),c(1,0),c(0,1),c(0,0)))))
)
```

Description

Geometric unary operations on simple feature geometries. 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 = 1, nQuadSegs = 30, endCapStyle = "ROUND",
    joinStyle = "ROUND", mitreLimit = 1)
```

```r
st_boundary(x)
```

```r
st_convex_hull(x)
```

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

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

```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)
```
st_point_on_surface(x)
st_node(x)

st_segmentize(x, dfMaxLength, ...)

Arguments

- **x**: object of class sf, sfg or sf
- **dist**: numeric; buffer distance for all, or for each of the elements in x; in case dist 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
- **endCapStyle**: character; style of line ends, one of 'ROUND', 'FLAT', 'SQUARE'
- **joinStyle**: character; style of line joins, one of 'ROUND', 'MITRE', 'BEVEL'
- **mitreLimit**: numeric; limit of extension for a join if joinStyle 'MITRE' is used (default 1.0, minimum 0.0)
- **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.
- **dTolerance**: numeric; tolerance parameter, specified for all or for each feature geometry.
- **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
- **...**: ignored
- **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 underlying 'buffer with style' GEOS function is used. See postgis.net/docs/ST_Buffer.html for details.
st_boundary returns the boundary of a geometry
st_convex_hull creates the convex hull of a set of points
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_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_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.

**Examples**

```r
## st_buffer, style options (taken from rgeos gBuffer)
ll = 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(ll, dist = 1, endCapStyle="ROUND"), reset = FALSE, main = "endCapStyle: ROUND")
plot(st_buffer(ll, dist = 1, endCapStyle="FLAT"), reset = FALSE, main = "endCapStyle: FLAT")
plot(st_buffer(ll, dist = 1, endCapStyle="SQUARE"), reset = FALSE, main = "endCapStyle: SQUARE")
plot(st_buffer(ll, dist = 1, nQuadSegs=1), reset = FALSE, main = "nQuadSegs: 1")
plot(st_buffer(ll, dist = 1, nQuadSegs=2), reset = FALSE, main = "nQuadSegs: 2")
plot(st_buffer(ll, dist = 1, nQuadSegs=5), reset = FALSE, main = "nQuadSegs: 5")
par(op)
```

```r
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(st_buffer(l2, dist = 1, joinStyle="MITRE"), reset = FALSE, main = "joinStyle: MITRE")
plot(st_buffer(l2, dist = 1, joinStyle="BEVEL"), reset = FALSE, main = "joinStyle: BEVEL")
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"))
plot(st_convex_hull(nc))
plot(nc, border = grey(.5))
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 (sf_extSoftVersion("GEOS") &gt; "3.5.0") {
  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(x, 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)
}
# 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:
pts$pols = pols[unlist(st_intersects(pts,pols))]
  plot(pts["id"], pch = 16) # ID is color
plot(st_set_geometry(pts, "pols")[["id"]], xlim = c(0,1), ylim = c(0,1), reset = FALSE)
plot(st_geometry(pts), add = TRUE)
}
mls = st_multilinestring(list(matrix(c(0,0,0,1,1,0,0),2,byrow=TRUE)))
st_polygonize(mls)
mls = st_multilinestring(list(rbind(c(0,0),c(1,1)), rbind(c(2,0), c(1,1))))
st_line_merge(st_sfc(mls))
plot(nc, axes = TRUE)
plot(st_centroid(nc), add = TRUE, pch = 3)
mp = st_combine(st_buffer(st_sfc(lapply(1:3, function(x) st_point(c(x,x)))), 0.2 * 1:3))
plot(mp)
plot(st_centroid(mp), add = TRUE, col = 'red') # centroid of combined geometry
plot(st_centroid(mp, of_largest_polygon = TRUE), add = TRUE, col = 'blue', pch = 3)
plot(nc, axes = TRUE)
plot(st_point_on_surface(nc), add = TRUE, pch = 3)
(l = st_linestring(rbind(c(0,0), c(1,1), c(0,1), c(1,0), c(0,0))))
st_polygonize(st_node(l))
st_node(st_multilinestring(list(rbind(c(0,0), c(1,1), c(0,1), c(1,0), c(0,0)))))))
sf = st_sf(a=1, geom=st_sfc(st_linestring(rbind(c(0,0),c(1,1))))), crs = 4326)
seg = st_segmentize(sf, units::set_units(100, km))
seg = st_segmentize(sf, units::set_units(0.01, rad))
nrow(seg$geom[[1]])
internal  

Internal functions

Description
Internal functions

Usage
.stop_geos(msg)

Arguments
  msg  error message

is_driver_available  Check if driver is available

Description
Search through the driver table if driver is listed

Usage
.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
.is_driver_can(drv, drivers = st_drivers(), operation = "write")
is_geometry_column

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

is_geometry_column(con, x, classes = "")

Arguments

con database connection
x inherits data.frame
classes classes inherited

merge.sf merge method for sf and data.frame object

Description

merge method for 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.

**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:
library(maps)
w = st_as_sf(map("world", plot = FALSE, fill = TRUE))
w2 = (st_geometry(w) + c(360,90)) %% c(360) - c(0,90)
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('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 = NULL, pal = NULL, nbreaks = 10,
     breaks = "pretty", max.plot = if (is.null(n <- options("sf_max.plot")[[1]]) 9 else n, key.pos = get_key_pos(x, ...),
     key.length = 0.618, key.width = lcm(1.8), reset = TRUE,
     logz = FALSE)
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)
```
## 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 overridden 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, or -1 to select automatically. If multiple columns are plotted in a single function call by default no key is plotted and every subplot 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 elements; if TRUE restore original mode after plotting; 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.

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)

rule
see polypath; for winding, exterior ring direction should be opposite that of the holes; with evenodd, plotting is robust against misspecified ring directions

xlim
see plot.window

ylim
see plot.window

asp
see below, and see par

axes
logical; should axes be plotted? (default FALSE)

bgc
background color
### 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 `m`row 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.

`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 `http://www.colorbrewer2.org/` (if `n < 9`, Set2, else Set3).
Examples

nc = st_read(system.file("pkg/nc.pkg", package="sf"), quiet = TRUE)
# plot single attribute, auto-legend:
plot(nc["SID74"])
# plot multiple:
plot(nc[cbind("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_centroid(st_geometry(nc)), col = 'red', add = TRUE)
# add a custom legend to an arbitray plot:
layout(matrix(1:2, ncol = 2), widths = c(1, lcm(2)))
plot()
.image_scale(1:10, col = sf.colors(9), key.length = lcm(8), key.pos = 4, at = 1:10)
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.

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

---

**sf** 

Create sf object

---

Description

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

Usage

```r
st_sf(..., agr = NA_agr_, row.names, stringsAsFactors = default.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: should character vectors be converted to factors? The 'factory-fresh' default is TRUE, but this can be changed by setting `options(stringsAsFactors = FALSE)`.
- `crs`: coordinate reference system: integer with the EPSG code, or character with `proj4string`
- `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 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
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,]
```


### Deprecated functions in sf

**Description**

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

**Usage**

```r
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`

---

### 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)

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

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()
sf_project

directly transform a set of coordinates

Description

directly transform a set of coordinates

Usage

sf_project(from, to, pts)

Arguments

from character; proj4string of pts
to character; target coordinate reference system(pts
two-column numeric matrix, or object that can be coerced into a matrix

sgbp

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.

st

Create simple feature from a numeric vector, matrix or list

Description

Create simple feature from a numeric vector, matrix or list

Usage

```r
st_point(x = c(NA_real_, NA_real_), dim = "XYZ")
st_multipoint(x = matrix(numeric(0), 0, 2), dim = "XYZ")
st_linestring(x = matrix(numeric(0), 0, 2), dim = "XYZ")
st_polygon(x = list(), dim = if (length(x)) "XYZ" else "XY")
st_multilinestring(x = list(), dim = if (length(x)) "XYZ" else "XY")
st_multipolygon(x = list(), dim = if (length(x)) "XYZ" else "XY")
st_geometrycollection(x = list(), dims = "XY")
```

## S3 method for class 'sfg'
print(x, ..., width = 0)

## S3 method for class 'sfg'
head(x, n = 10L, ...)

## S3 method for class 'sfg'
format(x, ..., width = 30)

## S3 method for class 'sfg'
c(..., recursive = FALSE, flatten = TRUE)

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

Arguments

- `x`: for `st_point`, numeric vector (or one-row-matrix) of length 2, 3 or 4; for `st_linestring` and `st_multipoint`, numeric matrix with points in rows; for `st_polygon` and `st_multilinestring`, list with numeric matrices with points in rows; for `st_multipolygon`, list of lists with numeric matrices; for `st_geometrycollection` list with (non-geometrycollection) simple feature objects.

- `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

```r
(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(pts))
outer = matrix(c(0,0,10,0,10,0,10,0,0,0), ncol=2, byrow=TRUE)
hole1 = matrix(c(1,1,1,2,2,2,1,1,1), ncol=2, byrow=TRUE)
hole2 = matrix(c(5,5,5,6,6,6,5,5,5), ncol=2, byrow=TRUE)
pts = list(outer, hole1, hole2)
(ml1 = st_multilinestring(pts))
pts3 = lapply(pts, function(x) cbind(x, 0))
(ml2 = st_multilinestring(pts3))
(ml3 = st_multilinestring(pts3, "XYM"))
pts4 = lapply(pts3, function(x) cbind(x, 0))
(ml4 = st_multilinestring(pts4))
outer = matrix(c(0,0,10,0,10,0,10,0,0,0), ncol=2, byrow=TRUE)
hole1 = matrix(c(1,1,1,2,2,2,1,1,1), ncol=2, byrow=TRUE)
hole2 = matrix(c(5,5,5,6,6,6,5,5,5), ncol=2, byrow=TRUE)
pts = list(outer, hole1, hole2)
(pl1 = st_polygon(pts))
pts3 = lapply(pts, function(x) cbind(x, 0))
(pl2 = st_polygon(pts3))
(pl3 = st_polygon(pts3, "XYM"))
pts4 = lapply(pts3, function(x) cbind(x, 0))
(pl4 = st_polygon(pts4))
pol1 = list(outer, hole1, hole2)
pol2 = list(outer + 12, hole1 + 12)
pol3 = list(outer + 24)
mp = list(pol1, pol2, pol3)
(mp1 = st_multipolygon(mp))
pts3 = lapply(mp, function(x) lapply(x, function(y) cbind(y, 0)))
(mp2 = st_multipolygon(pts3))
(mp3 = st_multipolygon(pts3, "XYM"))
pts4 = lapply(mp2, function(x) lapply(x, function(y) cbind(y, 0)))
(mp4 = st_multipolygon(pts4))
(gc = st_geometrycollection(list(pl, ls1, pl1, mp1)))
st_geometrycollection() # empty geometry
c(st_point(1:2), st_point(5:6))
c(st_point(1:2), st_multipoint(matrix(5:8,2)))
c(st_multipoint(matrix(1:4,2)), st_multipoint(matrix(5:8,2)))
c(st_linestring(matrix(1:6,3)), st_linestring(matrix(11:16,3)))
c(st_multilinestring(list(matrix(1:6,3))), st_multilinestring(list(matrix(11:16,3))))
pl = list(rbind(c(0,0), c(1,0), c(1,1), c(0,1), c(0,0)))
c(st_polygon(pl), st_polygon(pl))
c(st_polygon(pl), st_multipolygon(list(pl)))
c(st_linestring(matrix(1:6,3)), st_point(1:2))
c(st_linestring(list(st_point(1:2), st_linestring(matrix(1:6,3)))),
 st_multipolygon(list(st_multipoint(matrix(11:16,3)))))
c(st_multipoint(list(st_point(1:2), st_linestring(matrix(1:6,3)))),
 st_multipoint(list(matrix(11:16,3))), st_point(5:6),
 st_multipoint(list(st_point(10:11))));
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)`
- `.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)`
- `.image_scale_factor(z, col, breaks = NULL, 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
- `side` ignore
- `at` ignore
- `labels` ignore
- `...` ignore
- `lon` ignore
- `lat` ignore
- `ndiscr` ignore
- `reset` ignore
- `z` ignore
- `col` ignore
- `breaks` ignore
- `add.axis` ignore
- `axes` ignore
- `logz` ignore
- `key.width` ignore
**st_agr**

*get or set relation_to_geometry attribute of an sf object*

---

**Description**

get or set relation_to_geometry attribute of an sf object

**Usage**

```r
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 an 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, ..., EWKB = FALSE, pureR = FALSE, hex = FALSE)

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?

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](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**

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

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

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

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

## S3 method for class 'lpp'  

st_as_sf(x, ...)

**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

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.

**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_coordinates is FALSE.

**Examples**

pt1 = st_point(c(0,1))  
pt2 = st_point(c(1,1))  
st_sfc(pt1, pt2)  
d = data.frame(a = 1:2)  
d$geom = st_sfc(pt1, pt2)  
df = st_as_sf(d)  
d$geom = c("POINT(0 0)", "POINT(1 1)")  
df = st_as_sf(d, wkt = "geom")  
d$geom2 = st_sfc(pt1, pt2)  
st_as_sf(d)  
# should warn

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)  
library(sp)  
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
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 'list'
st_as_sfc(x, ..., crs = NA_crs)

## 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, ...)

st_as_sfc(x, ...)

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

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

Arguments

- **x**: object to convert
- **...**: further arguments
- **crs**: integer or character; coordinate reference system for the
- **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](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).
- **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

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("0101000002047010000000000801A06410000000AC5C1441"), class = "WKB")
st_as_sfc(wkb, EWKB = TRUE)
wkb = structure(list("0x01010000204701000000000801A06410000000AC5C1441"), 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(1663186 -105415,1664320 -104617)")
```

<table>
<thead>
<tr>
<th>st_as_text</th>
<th>Return Well-known Text representation of simple feature geometry or coordinate reference system</th>
</tr>
</thead>
</table>

Description

Return Well-known Text representation of simple feature geometry or coordinate reference system
48

Usage
---

```r
## S3 method for class 'crs'
st_as_text(x, ..., 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
- `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 SRID to a WKT string.

Examples
---

```r
st_as_text(st_point(l:2))
st_as_text(st_sfc(st_point(c(-90, 40)), crs = 4326), EWKT = TRUE)
```

---

**st_bbox**

*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_

### Arguments

- `x` object of class `bbox`
- `obj` object to compute the bounding box from
- `...` ignored
- `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))
```
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 '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

The `st_cast` method for `sf` objects can only split geometries, e.g. cast `MULTIPOINT` into multiple `POINT` 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

```
example(st_read)
mpl <- nc$geometry[[4]]
#st_cast(x) ## error 'argument "to" is missing, with no default'
cast_all <- function(xg) {
  lapply(c("MULTIPOINT", "MULTILINESTRING", "MULTIPOINT", "POLYGON", "LINESTRING", "POINT"),
    function(x) st_cast(xg, x))
```
Coerce geometry to MULTI* geometry

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

Usage
`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

\[
\text{st_collection_extract}(x, \text{type} = c(\text{"POLYGON"}, \text{"POINT"}, \text{"LINESTRING"}), \text{warn} = \text{FALSE})
\]

## S\text{3} method for class 'sfg'
\[
\text{st_collection_extract}(x, \text{type} = c(\text{"POLYGON"}, \text{"POINT"}, \text{"LINESTRING"}), \text{warn} = \text{FALSE})
\]

## S\text{3} method for class 'sfc'
\[
\text{st_collection_extract}(x, \text{type} = c(\text{"POLYGON"}, \text{"POINT"}, \text{"LINESTRING"}), \text{warn} = \text{FALSE})
\]

## S\text{3} method for class 'sf'
\[
\text{st_collection_extract}(x, \text{type} = c(\text{"POLYGON"}, \text{"POINT"}, \text{"LINESTRING"}), \text{warn} = \text{FALSE})
\]

Arguments

- \texttt{x} an object of class \texttt{sf}, \texttt{sfc} or \texttt{sfg} that has mixed geometry (GEOMETRY or GEOMETRYCOLLECTION).
- \texttt{type} character; one of "POLYGON", "POINT", "LINESTRING"
- \texttt{warn} logical; if \text{TRUE}, warn if attributes are assigned to sub-geometries when casting (see \texttt{st_cast})

Value

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

Examples

\[
\begin{align*}
\text{pt} &\leftarrow \text{st_point}(c(1, 0)) \\
\text{ls} &\leftarrow \text{st_linestring}(\text{matrix}(c(4, 3, 0, 0), \text{ncol} = 2)) \\
\text{poly1} &\leftarrow \text{st_polygon}(\text{list}(\text{matrix}(c(5.5, 7, 7, 6, 5.5, 0, 0, -0.5, -0.5, 0), \text{ncol} = 2))) \\
\text{poly2} &\leftarrow \text{st_polygon}(\text{list}(\text{matrix}(c(6.6, 8, 8, 7, 6.6, 1, 1.5, 1.5, 1), \text{ncol} = 2)))
\end{align*}
\]
st_coordinates <- 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  retrieve 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 \((X, Y, \text{possibly } Z \text{ and/or } M)\) in rows, possibly followed by integer indicators \(L_1, \ldots, L_3\) that point out to which structure the coordinate belongs; for POINT this is absent (each coordinate is a feature), for LINESTRING \(L_1\) refers to the feature, for MULTIPOLYGON \(L_1\) refers to the main ring or holes, \(L_2\) to the ring id in the MULTIPOLYGON, and \(L_3\) to the simple feature.

Description

crop an sf object to a specific rectangle

Usage

\[
\text{st\_crop}(x, y, \ldots)
\]

### S3 method for class 'sfc'

\[
\text{st\_crop}(x, y, \ldots, \text{xmin, ymin, xmax, ymax})
\]

### S3 method for class 'sf'

\[
\text{st\_crop}(x, y, \ldots)
\]

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))
```

### 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, proj4text = "", valid = TRUE, ...)
```

---

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

---

# 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
```
### S3 replacement method for class 'sfc'

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

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

```r
NA_crs_`
```

### S3 method for class 'crs'

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

### S3 method for class 'crs'

```r
x$name
```

#### Arguments

- `x` numeric, character, or object of class `sf` or `sfc`
- `...` ignored
- `proj4text` character. Must be used in conjunction with `valid = FALSE`.
- `valid` default `TRUE`. This allows to create `crs` without checking against the local `proj4` database. It can be used to synchronize `crs` with a remote database, but avoid it as much as possible.
- `wkt` character well-known-text representation of the `crs`
- `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 valid `proj4string` (ii) integer, a valid EPSG value (numeric), or (iii) a list containing named elements `proj4string` (character) and/or `epsg` (integer) with (i) and (ii).
- `name` element name; `epsg` or `proj4string`, or one of `proj4strings` named components without the `+`; see examples

#### 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 `epsg` (integer EPSG code) and `proj4string` (character). Two objects of class `crs` are semantically identical when: (1) they are completely identical, or (2) they have identical `proj4string` but one of them has a missing EPSG ID. As a consequence, equivalent but different `proj4strings`, e.g. "+proj=longlat +datum=WGS84" and "+datum=WGS84 +proj=longlat", are considered different. The operators `==` and `!=` are overloaded for `crs` objects to establish semantical identity.

In case a coordinate reference system is replaced, no transformation takes place and a warning is raised to stress this. EPSG values are either read from `proj4strings` that contain `+init=epsg:...` or set to 4326 in case the `proj4string` contains `+proj=longlat` and `+datum=WGS84`, literally.
If both epsg and proj4string are provided, they are assumed to be consistent. In processing them, the EPSG code, if not missing valued, is used and the proj4string is derived from it by a call to GDAL (which in turn will call PROJ.4). Warnings are raised when epsg is not consistent with a proj4string that is already present.

NA_crs_ is the crs object with missing values for epsg and proj4string.

**Value**

If x is numeric, return crs object for SRID x; if x is character, return crs object for proj4string x; if wkt is given, return crs object for well-known-text representation wkt; if x is of class sf or sfc, return its crs object.

Object of class crs, which is a list with elements epsg (length-1 integer) and proj4string (length-1 character).

**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)))
library(dplyr)
x = sfc %>% st_set_crs(4326) %>% st_transform(3857)
x
st_crs("+init=epsg:3857")$epsg
st_crs("+init=epsg:3857")$proj4string
st_crs("+init=epsg:3857 +units=m")$b # numeric
st_crs("+init=epsg:3857 +units=m")$units # character
```

---

**Description**

Get a list of the available GDAL drivers

**Usage**

```r
st_drivers(what = "vector")
```

**Arguments**

- what character: "vector" or "raster", anything else will return all drivers.
Details

The drivers available will depend on the installation of GDAL/OGR, and can vary: the `st_drivers()` function shows which are available, and which may be written (but all are assumed to be readable). Note that stray files in data source directories (such as *.dbf) may lead to spurious errors that accompanying *.shp are missing.

field vsi refers to the driver’s capability to read/create datasets through the VSI*L API.

Value

a data.frame with driver metadata

Examples

```r
st_drivers()
```
Arguments

- **obj**: object of class `sf` or `sfc`
- **...**: ignored
- **x**: object of class `data.frame`
- **value**: object of class `sfc`, or character

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`.

`st_drop_geometry` drops the geometry of its argument, and reclasses it accordingly.

Value

- `st_geometry` returns an object of class `sfc`, a list-column with geometries
- `st_geometry` returns an object of class `sfc`. Assigning geometry to a `data.frame` creates an `sf` object, assigning it to an `sf` object replaces the geometry list-column.

Examples

```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_geometry_type

**Return geometry type of an object**

Description

Return geometry type of an object, as a factor

Usage

```r
st_geometry_type(x)
```
Arguments

x  object of class sf or sfc

Value

a factor with the geometry type of each simple feature in x

---------
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; degrees east for the meridians

lat  numeric; degrees north for the parallels

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)
library(maps)

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[,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_graticule(4326), axes = TRUE, lon = seq(-60,-130,by=-10))
```

Description

Areal-weighted interpolation of polygon data
Usage

\texttt{st\_interpolate\_aw(x, to, extensive, ...)}

Arguments

- \texttt{x} \hspace{1cm} \text{object of class sf, for which we want to aggregate attributes}
- \texttt{to} \hspace{1cm} \text{object of class sf or sfc, with the target geometries}
- \texttt{extensive} \hspace{1cm} \text{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.}
- \texttt{...} \hspace{1cm} \text{ignored}

Examples

\begin{verbatim}
nc = st_read(system.file("shape/nc.shp", package="sf"))
g = st_make_grid(nc, n = c(20,10))
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)
\end{verbatim}

\begin{verbatim}
\texttt{st\_is} \hspace{1cm} \textit{test equality between the geometry type and a class or set of classes}
\end{verbatim}

Description

\textit{test equality between the geometry type and a class or set of classes}

Usage

\texttt{st\_is(x, type)}

Arguments

- \texttt{x} \hspace{1cm} \text{object of class sf, sfc or sfg}
- \texttt{type} \hspace{1cm} \text{character; class, or set of classes, to test against}
**st_is_longlat**

**Examples**

```r
st_is(st_point(0:1), "POINT")
sfc = st_sfc(st_point(0:1), st_linestring(matrix(1:6,2)))
st_is(sfc, "POINT")
st_is(sfc, "POLYGON")
st_is(sfc, "LINESTRING")
st_is(st_sf(a = 1:2, sfc), "LINESTRING")
st_is(sfc, c("POINT", "LINESTRING"))
```

**Description**

Assert whether simple feature coordinates are longlat degrees

**Usage**

```r
st_is_longlat(x)
```

**Arguments**

- `x` object of class `sf` or `sfc`

**Value**

TRUE if `+proj=longlat` is part of the proj4string, NA if this string is missing, FALSE otherwise

---

**st_jitter**

**jitter geometries**

**Description**

Jitter geometries

**Usage**

```r
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 \( \text{runif(1, -\text{amount}, \text{amount})} \) 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

```r
c = read_sf(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_j_metry(nc))
plot(st_jitter(st_geometry(nc), factor = .01), add = TRUE, col = '#ff8888')
```

Description

**spatial left or inner join**

Usage

```r
st_join(x, y, join = st_intersects, FUN, suffix = c(".x", ".y"). . . ,
left = TRUE, largest = FALSE)
```

Arguments

- `x` object of class `sf`
- `y` object of class `sf`
- `join` geometry predicate function with the same profile as `st_intersects`; see details
- `FUN` deprecated;
- `suffix` length 2 character vector; see `merge`
- `...` arguments passed on to the `join` function (e.g. `prepared`, or a pattern for `st_relate`)
- `left` logical; if TRUE carry out left join, else inner join; 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

Details

alternative values for argument `join` are: `st_disjoint st_touches st_crosses st_within st_contains st_overlaps st_covers st_covered_by st_equals` or `st_equals_exact`, or user-defined functions of the same profile
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)
library(dplyr)
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[1:10])[2:1], 1, paste,, collapse = " "),
    geom = st_make_grid(nc))
gr$col = sf.colors(10, categorical = TRUE, alpha = .3)
# cut, to check, NA's work out:
gr = gr[1:100,]
nc_j <- st_join(nc, gr, largest = TRUE)
# the two datasets:
opar <- par(mfrow = c(2,1), mar = rep(0,4))
plot(st_geometry(nc_j))
plot(st_geometry(gr), add = TRUE, col = gr$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_layers**

*List layers in a datasource*

**Description**

List layers in a datasource

**Usage**

```
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

---

**st_line_sample**

Sample points on a linear geometry

Description

Sample points on a linear geometry

Usage

```r
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"
- **sample**: numeric; a vector of numbers between 0 and 1 indicating the points to sample - if defined sample overrules n, density and type.

Examples

```r
ls = st_sfc(st_linestring(rbind(c(0,0),c(0,1))),
             st_linestring(rbind(c(0,0),c(10,0))))
st_line_sample(ls, density = 1)
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))
```
Create a regular tesselation over the bounding box of an sf or sfc object

**Arguments**

- **x**: object of class sf or sfc
- **cellsize**: target cell size
- **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

**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**

```r
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)
```
**st_nearest_feature**  
*get index of nearest feature*

**Description**
get index of nearest feature

**Usage**

```r
st_nearest_feature(x, y)
```

**Arguments**

- **x**: object of class `sfg`, `sfc` or `sf`
- **y**: object of class `sfg`, `sfc` or `sf`

**Value**
for each feature (geometry) in `x` the index of the nearest feature (geometry) in `y`

**See Also**

- `st_nearest_points` for finding the nearest points for pairs of feature geometries

**Examples**

```r
define function to calculate distance
def define function to calculate distance
def define function to calculate distance
def define function to calculate distance
def define function to calculate distance
```

```r
define function to calculate distance
def define function to calculate distance
def define function to calculate distance
def define function to calculate distance
def define function to calculate distance
```


```r
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 = 1:8, add = TRUE)
```

---

### st_nearest_points

**get nearest points between pairs of geometries**

**Description**

get nearest points between pairs of geometries

**Usage**

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

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

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

```r
## 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.

**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
r = sqrt(2)/10
tp1 = st_point(c(.1,.1))
tp2 = st_point(c(.9,.9))
tp3 = st_point(c(.9,.1))
b1 = st_buffer(pt1, r)
b2 = st_buffer(pt2, r)
b3 = st_buffer(pt3, r)
(1s0 = st_nearest_points(b1, b2)) # sfg
(1s = 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(is, add = TRUE, col = 'red')
```

```
nc = read_sf(system.file("gpkg/nc.gpkg", package="sf"))
plot(st_geometry(nc))
l1s = st_nearest_points(nc[1,], nc)
plot(l1s, col = 'red', add = TRUE)
pts = st_cast(l1s, "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**

`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)
```

---

st_precision  Get precision

Description

Get precision
Set precision

Usage

```r
st_precision(x)

st_set_precision(x, precision)

st_precision(x) <- value
```

Arguments

- `x` object of class `sfc` or `sf`
- `precision` numeric, or object of class `units` with distance units (but see details); see `st_as_binary` for how to do this.
- `value` precision value

Details

If `precision` is a `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 `sfc` object. The effect takes place in `st_binary` or, more precise, in the C++ function `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 `st_write` or `write_sf`, `st_make_valid` in package `lwgeom`; also `aggregate` and `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. `st_as_binary` contains an example of a roundtrip of `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

*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 = default.stringsAsFactors(), 
int64_as_string = FALSE, check_ring_dir = FALSE, 
fid_column_name = character())
```

```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 1d). 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.

**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? The ‘factory-fresh’ default is TRUE, but this can be changed by setting options(stringsAsFactors = FALSE).

**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)

**fid_column_name**
character; name of column to write feature IDs to; defaults to not doing this

**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.

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

**Examples**

```r
nc = st_read(system.file("shape/nc.shp", package="sf"))
summary(nc) # note that AREA was computed using Euclidian area on lon/lat degrees

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

## Not run:
  library(sp)
  example(meuse, ask = FALSE, echo = FALSE)
```
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**

```r
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 https://en.wikipedia.org/wiki/DE-9IM for further explanation.

Examples

```r
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 neighbors, 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") or an approximation of size (for other sampling types).

Usage
```
st_sample(x, size, ..., type = "random", exact = TRUE)
```
st_sample

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.
... ignored, or passed on to sample for multipoint sampling
type character; indicates the spatial sampling type; one of regular, hexagonal and regular.
exact logical; indicates the spatial sampling type; one of regular, hexagonal and regular.

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).

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)
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"))
plot(x2, graticule = g)
if (sf_extSoftVersion()["proj.4"] >= "4.9.0") {
p2 = st_transform(p, st_crs("+proj=ortho +lat_0=30 +lon_0=45"))
  plot(p2, add = TRUE)
}
x = st_sfc(st_polygon(list(rbind(c(0,0),c(90,0),c(90,10),c(0,90),c(0,0)))))) # NOT long/lat:
st_transform (x)
p_exact = st_sample(x, 1000, exact = TRUE)
p_not_exact = st_sample(x, 1000, exact = FALSE)
length(p_exact); length(p_not_exact)
plot(st_sample(x, 1000), add = TRUE)
x = st_sfc(st_polygon(list(rbind(c(-180, -90), c(180, -90), c(180, 90), c(-180, 90), c(-180, -90)))), crs = st_crs(4326))
if (sf_extSoftVersion()$"proj.4" >= "4.9.0") {
P = st_sample(x, 1000)
  st_sample(p, 3)
} # hexagonal:
sfc = st_sfc(st_polygon(list(rbind(c(0, 0), c(1, 0), c(1, 1), c(0, 0)))))
plot(sfc)
h = st_sample(sfc, 100, type = "hexagonal")
h1 = st_sample(sfc, 100, type = "hexagonal")
plot(h, add = TRUE)
plot(h1, col = 'red', add = TRUE)
c(length(h), length(h1)) # approximate!
pt = st_multipoint(matrix(1:20, 2))
ls = st_sfc(st_linestring(rbind(c(0, 0), c(0, 1))),
              st_linestring(rbind(c(0, 0), c(1, 0))),
              st_linestring(rbind(c(0, 1), c(1, 1))),
              st_linestring(rbind(c(2, 2), c(2, 0.00001))))
st_sample(ls, 80)
plot(st_sample(ls, 80))

---

st_transform

Transform or convert coordinates of simple feature

Description

Transform or convert coordinates of simple feature

Usage

st_transform(x, crs, ...)

## S3 method for class 'sfc'
st_transform(x, crs, ..., partial = TRUE, check = FALSE,
             use_gdal = TRUE)

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

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

st_proj_info(type = "proj")
Arguments

- **x**: object of class sf, sfc or sfg
- **crs**: coordinate reference system: integer with the EPSG code, or character with proj4string
- **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; perform a sanity check on resulting polygons?
- **use_gdal**: logical; this parameter is deprecated. For transformations using PROJ.4 directly rather than indirectly through GDAL, use `st_transform_proj` of package lwgeom (see Details)
- **type**: character; one of have_datum_files, proj, ellps, datum, units or prime_meridians; see Details.
- **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?

Details

Transforms coordinates of object to new projection. Features that cannot be transformed are returned as empty geometries.

`st_transform` uses GDAL for coordinate transformations; internally, GDAL converts the proj4string into a well-known-text representation, before passing that on to PROJ.4. In this process, some information can get lost. Adding parameter `+wktext` to the proj4string definition may resolve this; see https://github.com/edzer/sp/issues/42.

Some PROJ.4 projections are not supported by GDAL, e.g. "+proj=wintri" because it does not have an inverse projection. Projecting to unsupported projections can be done by `st_transform_proj`, part of package lwgeom. Note that the unsupported proj4string cannot be passed as argument to `st_crs`, but has to be given as character string.
The `st_transform` method for `sfg` objects assumes that the CRS of the object is available as an attribute of that name.

`st_proj_info` lists the available projections, ellipses, datums or units supported by the Proj.4 library when type is equal to proj, ellps, datum or units; when type equals have datum files a boolean is returned indicating whether datum files are installed and accessible (checking for conus).

PROJ >= 6 does not provide option type = "datums". PROJ < 6 does not provide the option type = "prime meridians".

For a discussion of using options, see https://github.com/r-spatial/sf/issues/280 and https://github.com/r-spatial/sf/issues/541

**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)
nc = st_read(system.file("shape/nc.shp", package="sf"))
st_area(nc[,]) # area from long/lat

st_area(st_transform(nc[,], 32191)) # NC state plane, m
st_area(st_transform(nc[,], 2264)) # NC state plane, US foot library(units)

set_units(st_area(st_transform(nc[,], 2264)), m^2)

st_transform(structure(p1, proj4string = "+init=epsg:4326"), "+init=epsg:3857")

st_proj_info("datum")

st_wrap_dateline(st_sfc(st_linestring(rbind(c(-179,0),c(179,0))), crs = 4326))
library(maps)
wrl <- st_as_sf(maps::map("world", fill = TRUE, plot = FALSE))
wrl_wrap <- st_wrap_dateline(wrl, options = c("WRAPDATELINE=YES", "DATELINEOFFSET=180"), quiet = TRUE)
wrl_moll <- st_transform(wrl_wrap, "+proj=moll")
plot(st_geometry(wrl_moll), col = "transparent")
```

---

**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)
```
st_write

Write simple features object to file or database

Arguments

- **x** object of class sf, sfc or sfg object
- **...** parameters passed on to viewport
- **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')))))
```

Description

Write simple features object to file or database

Usage

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

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

## S3 method for class 'sf'
```r
st_write(obj, dsn, layer = NULL, ..., 
driver = guess_driver_can_write(dsn), dataset_options = NULL, 
layer_options = NULL, quiet = FALSE, factorsAsCharacter = TRUE, 
update = driver %in% db_drivers, delete_dsn = FALSE, 
delete_layer = FALSE, fid_column_name = NULL)
```
## S3 method for class 'data.frame'

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

```r
write_sf(..., quiet = TRUE, delete_layer = TRUE)
```

## S4 method for signature 'PostgreSQLConnection,character,sf'

```r
dbWriteTable(conn, name, value, ..., row.names = FALSE, overwrite = FALSE, append = FALSE,
            field.types = NULL, factorsAsCharacter = TRUE, binary = TRUE)
```

## S4 method for signature 'DBIOobject,character,sf'

```r
dbWriteTable(conn, name, value, ..., row.names = FALSE, overwrite = FALSE, append = FALSE,
            field.types = NULL, factorsAsCharacter = TRUE, binary = TRUE)
```

### Arguments

- **obj**
  - object of class `sf` or `sfc`

- **dsn**
  - data source name (interpretation varies by driver - for some drivers, `dsn` is a file name, but may also be a folder or contain a database name) or a Database Connection (currently official support is for RPostgreSQL connections)

- **layer**
  - layer name (varies by driver, may be a file name without extension); 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 [http://www.gdal.org/ogr_formats.html](http://www.gdal.org/ogr_formats.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 objects into character strings (default), else into numbers by `as.numeric`.

- **update**
  - logical; FALSE by default for single-layer drivers but TRUE by default for database drivers as defined by `db_drivers`. For database-type drivers (e.g. GPKG) TRUE values will make GDAL try to update (append to) the existing data source, e.g. adding a table to an existing database.

- **delete_dsn**
  - logical; delete data source `dsn` before attempting to write?

- **delete_layer**
  - logical; delete layer `layer` before attempting to write? (not yet implemented)

- **fid_column_name**
  - character, name of column with feature IDs; if specified, this column is no longer written as feature attribute.
**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).

**Details**

Columns (variables) of a class not supported are dropped with a warning. When deleting layers or data sources is not successful, no error is emitted. `delete_dsn` and `delete_layers` should be handled with care; the former may erase complete directories or databases.

**Value**

obj, invisibly; in case obj is of class sfc, it is returned as an sf object.

**See Also**

st_drivers

**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
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=postgresql", "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**

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

**Arguments**

- `x`: object of class `sfg`, `sfc` or `sf`
- `...`: ignored
- `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)
```

### summary.sfc

**Summarize simple feature column**

**Description**

Summarize simple feature column.

**Usage**

```r
## S3 method for class 'sfc'
summary(object, ..., maxsum = 7L, maxpTs = 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.4 string

tibble Summarize simple feature type for tibble

Description

Summarize simple feature type for tibble
Summarize simple feature item for tibble

Usage

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

filter.sf(.data, ..., .dots)
arrange.sf(.data, ..., .dots)
group_by.sf(.data, ..., add = FALSE)
ungroup.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)
distinct.sf(.data, ..., .keep_all = FALSE)
gather.sf(data, key, value, ..., na.rm = FALSE, convert = FALSE,
factor_key = FALSE)
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, ..., .key = "data")
separate.sf(data, col, into, sep = "^[[:alnum:]]+", remove = TRUE,
convert = FALSE, extra = "warn", fill = "warn", ...)
unite.sf(data, col, ..., sep = "+", remove = TRUE)
unnest.sf(data, ..., .preserve = NULL)
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
.social 
.data 
... other arguments
.dots see corresponding function in package dplyr
.add see corresponding function in dplyr
.x tbls to join
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.
.keep_all see corresponding function in dplyr
data see original function docs
.key see original function docs
.value see original function docs
.na.rm see original function docs
.convert see original function docs
.factor_key see original function docs
.fill see original function docs
.drop see original function docs
.sep see original function docs
tbl see original function docs
.size see original function docs
.replace see original function docs
.weight see original function docs
.env see original function docs
.key see nest
col see separate
into see separate
remove see separate
extra see separate
.preserve see unnest

y tbls to join
by a character vector of variables to join by. If NULL, the default, *_.join() will do a natural join, using all variables with common names across the two tables. A message lists the variables so that you can check they’re right (to suppress the message, simply explicitly list the variables that you want to join).

To join by different variables on x and y use a named vector. For example, by = `c("a" = "b")` will match x.a to y.b.

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.

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

Value

an object of class `sf`

Examples

library(dplyr)
nc = st_read(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")
nc$area_cl = cut(nc$AREA, c(0, .1, .12, .15, .25))
nc %>% group_by(area_cl) %>% class()
nc2 <- nc %>% mutate(area10 = AREA/10)
nc %>% transmute(\(\text{AREA} = \text{AREA}/10\), \text{geometry} = \text{geometry}) %>% class()
nc %>% transmute(\(\text{AREA} = \text{AREA}/10\)) %>% class()
nc %>% select(SID74, SID79) %>% names()
nc %>% select(SID74, SID79, \text{geometry}) %>% names()
nc %>% select(SID74, SID79) %>% class()
nc %>% select(SID74, SID79, \text{geometry}) %>% class()
nc2 <- nc %>% rename(area = AREA)
nc %>% slice(1:2)
nc$area_cl = cut(nc$AREA, c(0, .1, .12, .15, .25))
nc.g <- nc %>% group_by(area_cl)
nc.g %>% summarise(mean(\(\text{AREA}\)))
nc.g %>% summarise(mean(\(\text{AREA}\))) %>% plot(col = grey(3:6 / 7))
nc[c(1:100, 1:10), ] %>% distinct() %>% nrow()
library(tidyrr)
nc %>% select(SID74, SID79) %>% gather("VAR", "SID", \text{geometry}) %>% summary()
library(tidyrr)
nc$row = 1:100 # needed for spread to work
nc %>% select(SID74, SID79, \text{geometry}, row) %>%
gather("VAR", "SID", \text{geometry}, row) %>%
spread(VAR, SID) %>% head()
storms.sf = st_as_sf(storms, coords = c("long", "lat"), crs = 4326)
x <- storms.sf %>% group_by(name, year) %>% nest
trs = lapply(x$data, function(tr) st_cast(st_combine(tr), "LINESTRING")[[1]]) %>% st_sfc(crs = 4326)
trs.sf = st_sf(x[,1:2], trs)
plot(trs.sf["year"], axes = TRUE)
Index

*Topic datasets
  db_drivers, 8
  extension_map, 8
  prefix_map, 31
  st_agr, 41
  st_bbox, 48
  st_crs, 57

*Topic data
  nc, 26
  .degAxis (stars), 40
  .get_layout (stars), 40
  .image_scale (stars), 40
  .image_scale_factor (stars), 40
  .stop_geos (internal), 24
  [.data_frame, 32, 33
  [.sf (sf)], 32
  $ .crs (st_crs), 57

  aggregate, 4, 52, 73
  aggregate (aggregate.sf), 4
  aggregate.sf, 4
  anti_join.sf (tidyverse), 87
  arrange.sf (tidyverse), 87
  as, 5
  as.data.frame, 87
  as.matrix.sfg (st), 37
  as.matrix.sgbp (sgbp), 36
  as_Spatial (as), 5
  as_Spatial(), 5

  basename, 84
  bind, 6
  bind_cols, 7
  bpy.colors, 30

  c, 16
  c.sfg, 4, 16, 90
  c.sfg (st), 37
  cbind, 7
  cbind.sf (bind), 6

  classIntervals, 29
  coerce, sf, Spatial-method (as), 5
  coerce, sfc, Spatial-method (as), 5
  coerce, Spatial, sf-method (as), 5
  coerce, Spatial, sfc-method (as), 5

  data.frame, 7
  db_drivers, 8
  dbDataType, DBIObject, sf-method
    (dbDataType, PostgreSQLConnection, sf-method), 7
  dbDataType, PostgreSQLConnection, sf-method, 7
  dbWriteTable, 84
  dbWriteTable, DBIObject, character, sf-method
    (st_write), 83
  dbWriteTable, PostgreSQLConnection, character, sf-method
    (st_write), 83
  dim.sgbp (sgbp), 36
  distinct.sf (tidyverse), 87
  dotsMethods, 6

  extension_map, 8

  filter.sf (tidyverse), 87
  format.sfg (st), 37
  full_join.sf (tidyverse), 87

  gather.sf (tidyverse), 87
  gdal, 9
  gdal_crs (gdal), 9
  gdal_inv_geotransform (gdal), 9
  gdal_metadata (gdal), 9
  gdal_polygonize (gdal), 9
  gdal_rasterize (gdal), 9
  gdal_read (gdal), 9
  gdal_subdatasets (gdal), 9
  gdal_utils, 11
  gdal_write (gdal), 9
  geos_binary_ops, 12
INDEX

geos_binary_pred, 14
geos_combine, 16
geos_measures, 17
geos_query, 19
geos_unary, 20
get_key_pos(plot), 27
getwd, 76
group_by.sf(tidyverse), 87
gUnaryUnion, 16
gUnion, 16

head.sfg(st), 37
Id, 75
inner_join.sf(tidyverse), 87
internal, 24
intersect, 13
is.na.bbox(st_bbox), 48
is.na.crs(st_crs), 57
is_driver_available, 24
is_driver_can, 24
is_geometry_column, 25

left_join, 66
left_join.sf(tidyverse), 87
merge, 66
merge.sf, 25
mutate.sf(tidyverse), 87

NA_agr_(st_agr), 41
NA_bbox_(st_bbox), 48
NA_crs_(st_crs), 57
nc, 26
nest, 89
nest.sf(tidyverse), 87

obj_sum.sfc(tibble), 87
Ops, 26

par, 29, 30
pillar_shaft.sfc(tibble), 87
plot, 27, 29
plot.window, 29
plot.sf, 29
plot_sf(plot), 27
polypath, 29
prefix_map, 31
print.sf(sf), 32
print.sfg(st), 37

print.sgbp(sgbp), 36
rainbow, 29
rawToHex, 31
rbind, 6
rbind.sf(bind), 6
read_sf(st_read), 74
rename.sf(tidyverse), 87
right_join.sf(tidyverse), 87

sample, 79
sample_frac.sf(tidyverse), 87
sample_n.sf(tidyverse), 87
select.sf(tidyverse), 87
semi_join.sf(tidyverse), 87
separate, 89, 90
separate.sf(tidyverse), 87
set_units, 18
setdiff, 13
sf, 4, 32, 58, 61, 62, 65, 69, 76, 89, 90
sf-deprecated, 34
sf-deprecated-package(sf-deprecated), 34
sf.colors, 29
sf.colors(plot), 27
sf_extSoftVersion, 35
sf_project, 36
sfc, 34, 44, 58, 61, 62, 65, 69, 71
gbp, 15, 36
slice.sf(tidyverse), 87
sp, 6
spread.sf(tidyverse), 87
st, 37
st_agr, 41
st_agr<-(st_agr), 41
st_as_binary, 32, 35, 41, 47, 73, 74
st_as_grob, 43
st_as_sf, 34, 43, 75
st_as_sfc, 32, 33, 45, 50
st_as_text, 47
st_bbox, 48, 56
st_bind_cols(bind), 6
st_boundary(geos_unary), 20
st_buffer(geos_unary), 20
st_cast, 18, 51, 54
st_cast(), 5
st_cast_sfc_default, 53
st_centroid(geos_unary), 20
st_collection_extract, 54
st_combine, 89
st_combine (geos_combine), 16
st_contains, 66
st_contains (geos_binary_pred), 14
st_contains_properly
  (geos_binary_pred), 14
st_convex_hull (geos_binary_pred), 20
st_coordinates, 55
st_covered_by, 66
st_covered_by (geos_binary_pred), 14
st_covers, 66
st_covers (geos_binary_pred), 14
st_crop, 56
st_crosses, 66
st_crosses (geos_binary_pred), 14
st_crs, 10, 50, 57, 81
st_crs().47
st_crs<-(st_crs), 57
st_difference, 17
st_difference (geos_binary_ops), 12
st_dimension, 18
st_dimension (geos_query), 19
st_disjoint, 66
st_disjoint (geos_binary_pred), 14
st_distance (geos_measures), 17
st_drivers, 24, 25, 59, 85
st_drop_geometry (st_geometry), 60
st_equals, 66, 90
st_equals (geos_binary_pred), 14
st_equals_exact, 66
st_equals_exact (geos_binary_pred), 14
st_geod_area, 18
st_geod_segmentize, 21
st_geometry, 60
st_geometry<-(st_geometry), 60
st_geometry_type, 61
st_geometrycollection (st), 37
st_graticule, 30, 62
st_interpolate_aw, 63
st_intersection, 15, 17, 78
st_intersection (geos_binary_ops), 12
st_intersects, 13, 42, 66
st_intersects (geos_binary_pred), 14
st_is, 64
st_is_empty (geos_query), 19
st_is_longlat, 65
st_is_simple (geos_query), 19
st_is_valid (geos_query), 19
st_is_within_distance
  (geos_binary_pred), 14
st_jitter, 65
st_join, 4, 66
st_layers, 67, 76
st_length (geos_measures), 17
st_line_merge (geos_unary), 20
st_line_sample, 68
st_linestring (st), 37
st_make_grid, 69
st_multilinestring (st), 37
st_multipoint (st), 37
st_multipolygon (st), 37
st_nearest_feature, 70, 71
st_nearest_points, 70, 71
st_node (geos_unary), 20
st_normalize, 72
st_overlaps, 66
st_overlaps (geos_binary_pred), 14
st_point, 44
st_point (st), 37
st_point_on_surface (geos_unary), 20
st_polygon (st), 37
st_polygonize (geos_unary), 20
st_precision, 73
st_precision<-(st_precision), 73
st_proj_info (st_transform), 80
st_read, 32, 34, 35, 74
st_read_db (sf-deprecated), 34
st_read_db, (sf-deprecated), 34
st_relate, 15, 66, 77
st_sample, 78
st_segmentize (geos_unary), 20
st_set_agr (st_agr), 41
st_set_crs (st_crs), 57
st_set_geometry (st_geometry), 60
st_set_precision (st_precision), 73
st_sf, 6, 7, 44
st_sf (sf), 32
st_sfc (sfc), 34
st_simplify (geos_unary), 20
st_snap (geos_binary_ops), 12
st_sym_difference, 17
st_sym_difference (geos_binary_ops), 12
st_touches, 66
st_touches (geos_binary_pred), 14
st_transform, 80
INDEX

st_transform_proj, 81
st_triangulate (geos_unary), 20
st_union, 4, 13, 89
st_union (geos_combine), 16
st_viewport, 82
st_voronoi (geos_unary), 20
st_within, 66
st_within (geos_binary_pred), 14
st_wrap_dateline (st_transform), 80
st_write, 34, 73, 83
st_write_db (sf-deprecated), 34
st_zm, 86
st_zm(), 6
stars, 40
summarise, 52, 73
summarise (tidyverse), 87
summary.sfc, 86
t.sgbp (sgbp), 36
tibble, 87
tidyverse, 87
transmute.sf (tidyverse), 87
type_sum, 87
type_sum.sfc (tibble), 87
ungroup.sf (tidyverse), 87
unite.sf (tidyverse), 87
unnest, 90
unnest.sf (tidyverse), 87
viewport, 83
write_sf, 73
write_sf (st_write), 83