

# Package ‘sf’

October 31, 2017

**Version** 0.5-5

**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.4 for projection conversions and datum transformations.

**Depends** R (>= 3.3.0), methods

**Imports** utils, stats, tools, graphics, grDevices, grid, Rcpp, DBI (>= 0.5), units (>= 0.4-6), classInt, magrittr

**Suggests** geosphere (>= 1.5-5), maptools, maps, rgdal, rgeos, sp (>= 1.2-4), raster, RSQLite, RPostgreSQL, tibble, rlang, dplyr (>= 0.7-0), tidyr (>= 0.7-0), ggplot2, mapview, tmap, testthat, knitr, covr, microbenchmark, spatstat

**LinkingTo** Rcpp

**VignetteBuilder** knitr

**SystemRequirements** GDAL (>= 2.0.0), GEOS (>= 3.3.0), PROJ.4 (>= 4.8.0)

**License** GPL-2 | MIT + file LICENSE

**URL** <https://github.com/r-spatial/sf/>

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

**Collate** RcppExports.R init.R crs.R bbox.R read.R db.R sfc.R sfg.R sf.R  
bind.R wkb.R wkt.R plot.R geom.R transform.R sp.R grid.R  
arith.R tidyverse.R cast\_sfg.R cast\_sfc.R graticule.R  
datasets.R aggregate.R agr.R maps.R join.R sample.R valid.R  
geohash.R split.R collection\_extract.R jitter.R sgbp.R  
spatstat.R

**RoxygenNote** 6.0.1

**NeedsCompilation** yes

**Author** Edzer Pebesma [aut, cre] (0000-0001-8049-7069),  
Roger Bivand [ctb] (0000-0003-2392-6140),  
Ian Cook [ctb],  
Tim Keitt [ctb],

Michael Sumner [ctb],  
 Robin Lovelace [ctb],  
 Hadley Wickham [ctb],  
 Jeroen Ooms [ctb],  
 Etienne Racine [ctb]

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

**Repository** CRAN

**Date/Publication** 2017-10-31 16:03:27 UTC

## R topics documented:

aggregate.sf . . . . .	3
as . . . . .	4
bgMap . . . . .	5
bind . . . . .	5
db_drivers . . . . .	6
dplyr . . . . .	7
extension_map . . . . .	10
geos_binary_ops . . . . .	10
geos_binary_pred . . . . .	11
geos_combine . . . . .	13
geos_measures . . . . .	14
geos_query . . . . .	16
geos_unary . . . . .	17
is_driver_available . . . . .	19
is_driver_can . . . . .	19
merge.sf . . . . .	20
Ops.sfg . . . . .	20
plot . . . . .	21
prefix_map . . . . .	25
rawToHex . . . . .	26
sf . . . . .	26
sfc . . . . .	28
sf_extSoftVersion . . . . .	28
srgb . . . . .	29
st . . . . .	29
st_agr . . . . .	32
st_as_binary . . . . .	33
st_as_grob . . . . .	34
st_as_sf . . . . .	34
st_as_sfc . . . . .	36
st_as_text . . . . .	38
st_bbox . . . . .	39
st_cast . . . . .	41
st_cast_sfc_default . . . . .	43
st_collection_extract . . . . .	44
st_coordinates . . . . .	45

st_crs . . . . .	46
st_drivers . . . . .	48
st_geohash . . . . .	49
st_geometry . . . . .	50
st_geometry_type . . . . .	51
st_graticule . . . . .	52
st_interpolate_aw . . . . .	53
st_is . . . . .	54
st_is_longlat . . . . .	55
st_jitter . . . . .	55
st_join . . . . .	56
st_layers . . . . .	57
st_line_sample . . . . .	57
st_make_grid . . . . .	58
st_precision . . . . .	59
st_read . . . . .	60
st_relate . . . . .	62
st_sample . . . . .	63
st_split . . . . .	64
st_transform . . . . .	65
st_viewport . . . . .	67
st_write . . . . .	68
st_zm . . . . .	70
summary.sfc . . . . .	71
tibble . . . . .	71
valid . . . . .	72

**Index****73**


---

aggregate.sf	<i>aggregate an sf object</i>
--------------	-------------------------------

---

**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 <a href="#">sf</a>
by	either a list of grouping elements, each as long as the variables in the data frame x (see <a href="#">aggregate</a> ), or an object of class sf or sfc, the geometries of which are used to find aggregation groups of x by using function join
FUN	function passed on to <a href="#">aggregate</a> , 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 <a href="#">st_union</a> ?
simplify	logical; see <a href="#">aggregate</a>
join	logical spatial predicate function to use if by is a simple features object or geometry; see <a href="#">st_join</a>

**Value**

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

**Examples**

```

m1 = cbind(c(0, 0, 1, 0), c(0, 1, 1, 0))
m2 = cbind(c(0, 1, 1, 0), c(0, 0, 1, 0))
pol = st_sfc(st_polygon(list(m1)), st_polygon(list(m2)))
set.seed(1985)
d = data.frame(matrix(runif(15), ncol = 3))
p = st_as_sf(x = d, coords = 1:2)
plot(pol)
plot(p, add = TRUE)
(p_ag1 = aggregate(p, pol, mean))
plot(p_ag1) # geometry same as pol
# works when x overlaps multiple objects in 'by':
p_buff = st_buffer(p, 0.2)
plot(p_buff, add = TRUE)
(p_ag2 = aggregate(p_buff, pol, mean)) # increased mean of second
# with non-matching features
m3 = cbind(c(0, 0, -0.1, 0), c(0, 0.1, 0.1, 0))
pol = st_sfc(st_polygon(list(m3)), st_polygon(list(m1)), st_polygon(list(m2)))
(p_ag3 = aggregate(p, pol, mean))
plot(p_ag3)

```

---

as *Methods to coerce simple feature geometries to corresponding Spatial\* objects*

---

**Description**

Methods to coerce simple feature geometries to corresponding Spatial\* objects

**Usage**

```
as_Spatial(from, cast = TRUE, IDs = paste0("ID", 1:length(from)))
```

**Arguments**

from	object of class <code>sfc_POINT</code> , <code>sfc_MULTIPPOINT</code> , <code>sfc_LINestring</code> , <code>sfc_MULTILINESTRING</code> , <code>sfc_POLYGON</code> , or <code>sfc_MULTIPOLYGON</code> .
cast	logical; if TRUE, <code>st_cast</code> from before converting, so that e.g. <code>GEOMETRY</code> objects with a mix of <code>POLYGON</code> and <code>MULTIPOLYGON</code> are cast to <code>MULTIPOLYGON</code> .
IDs	character vector with IDs for the <code>Spatial*</code> geometries

**Value**

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

**Examples**

```
nc = st_read(system.file("shape/nc.shp", package="sf"))
as_Spatial(st_geometry(nc[1,]))
```

---

bgMap

*This is data included in sf*

---

**Description**

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

---

 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.

---

dplyr

*Dplyr verb methods for sf objects*

---

**Description**

Dplyr verb methods for sf objects. Geometries are sticky, use [as.data.frame](#) to let dplyr's own methods drop them.

**Usage**

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

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

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

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

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

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

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

```
select.sf(.data, ...)
```

```
rename.sf(.data, ..., .dots)
```

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

```
summarise.sf(.data, ..., .dots, do_union = TRUE)
```

```
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)

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, ...)

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

```

### Arguments

<code>.data</code>	data object of class <code>sf</code>
<code>...</code>	other arguments
<code>.dots</code>	see corresponding function in package <code>dplyr</code>
<code>.keep_all</code>	see corresponding function in <code>dplyr</code>
<code>add</code>	see corresponding function in <code>dplyr</code>
<code>x</code>	see <a href="#">left_join</a>
<code>do_union</code>	logical; should geometries be unioned by using <a href="#">st_union</a> , or simply be combined using <a href="#">st_combine</a> ? Using <a href="#">st_union</a> resolves internal boundaries, but in case of unioning points may also change the order of the points.
<code>data</code>	see original function docs
<code>key</code>	see original function docs
<code>value</code>	see original function docs
<code>na.rm</code>	see original function docs
<code>convert</code>	see original function docs
<code>factor_key</code>	see original function docs
<code>fill</code>	see original function docs
<code>drop</code>	see original function docs
<code>sep</code>	see original function docs
<code>tbl</code>	see original function docs
<code>size</code>	see original function docs
<code>replace</code>	see original function docs



weight	see original function docs
.env	see original function docs
.key	see <a href="#">nest</a>
col	see <a href="#">separate</a>
into	see <a href="#">separate</a>
remove	see <a href="#">separate</a>
extra	see <a href="#">separate</a>
y	see <a href="#">left_join</a>
by	see <a href="#">left_join</a>
copy	see <a href="#">left_join</a>
suffix	see <a href="#">left_join</a>

## 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.

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

## 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[c(1:100, 1:10), ] %>% distinct() %>% nrow()
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(AREA = AREA/10, geometry = geometry) %>% class()
nc %>% transmute(AREA = AREA/10) %>% class()
nc %>% select(SID74, SID79) %>% names()
nc %>% select(SID74, SID79, geometry) %>% names()
nc %>% select(SID74, SID79) %>% class()
nc %>% select(SID74, SID79, 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(AREA))
nc.g %>% summarise(mean(AREA)) %>% plot(col = grey(3:6 / 7))
nc %>% as.data.frame %>% summarise(mean(AREA))
library(tidyr)
nc %>% select(SID74, SID79, geometry) %>% gather(VAR, SID, -geometry) %>% summary()
library(tidyr)
```

```

nc$row = 1:100 # needed for spread to work
nc %>% select(SID74, SID79, geometry, row) %>%
gather(VAR, SID, -geometry, -row) %>%
spread(VAR, SID) %>% head()
storms.sf = st_as_sf(storms, coords = c("long", "lat"), crs = 4326)
x <- storms.sf %>% group_by(name, year) %>% nest
trs = lapply(x$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)

```

---

extension_map	<i>Map extension to driver</i>
---------------	--------------------------------

---

### Description

Map extension to driver

### Usage

```
extension_map
```

### Format

An object of class list of length 24.

---

geos_binary_ops	<i>Geometric operations on pairs of simple feature geometry sets</i>
-----------------	--

---

### Description

Perform geometric set operations with simple feature geometry collections

### Usage

```
st_intersection(x, y)
```

```
st_difference(x, y)
```

```
st_sym_difference(x, y)
```

### Arguments

x                    object of class sf, sfc or sfg

y                    object of class sf, sfc or sfg

**Details**

A spatial index is built on argument `x`; see <http://r-spatial.org/r/2017/06/22/spatial-index.html>. The reference for the STR tree algorithm is: Leutenegger, Scott T., Mario A. Lopez, and Jeffrey Edgington. "STR: A simple and efficient algorithm for R-tree packing." Data Engineering, 1997. Proceedings. 13th international conference on. IEEE, 1997. For the pdf, search Google Scholar.

**Value**

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

**See Also**

[st\\_union](#) for the union of simple features collections; [intersect](#) and [setdiff](#) for the base R set operations.

**Examples**

```
# A helper function that erases all of y from x:
st_erase = function(x, y) st_difference(x, st_union(st_combine(y)))
```

---

geos\_binary\_pred

*Geometric binary predicates on pairs of simple feature geometry sets*

---

**Description**

Geometric binary predicates on pairs of simple feature geometry sets

**Usage**

```
st_intersects(x, y, sparse = TRUE, prepared = 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, prepared = FALSE)

```

### Arguments

x	object of class sf, sfc or sfg
y	object of class sf, sfc or sfg
sparse	logical; should a sparse index list be returned (TRUE) or a dense logical matrix? See below.
prepared	logical; prepare geometry for x, before looping over y?
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

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.

Sparse geometry binary predicate (sgbp) lists have the following attributes: `region.id` with the `row.names` of x (if any, else 1:n), 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.

`st_is_within_distance` returns a sparse matrix only, and can only be used for non-geographic (Cartesian) coordinates; use `st_distance(x,y) <= dist` to obtain the corresponding dense logical matrix.

**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 `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` a numeric vector with the 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.

**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, with or without resolving internal boundaries

**Usage**

```
st_combine(x)

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

**Arguments**

<code>x</code>	object of class <code>sf</code> , <code>sfc</code> or <code>sfg</code>
<code>y</code>	object of class <code>sf</code> , <code>sfc</code> or <code>sfg</code> (optional)
<code>...</code>	ignored
<code>by_feature</code>	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.

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, the area of a geometry, or the length of a geometry.

**Usage**

```
st_area(x)
```

```
st_length(x, dist_fun = geosphere::distGeo)
```

```
st_distance(x, y, dist_fun, by_element = FALSE)
```

**Arguments**

x	object of class sf, sfc or sfg
dist_fun	function to be used for great circle distances of geographical coordinates; for unprojected (long/lat) data, this should be a distance function of package <code>geosphere</code> , or compatible to that; it defaults to <code>distGeo</code> in that case; for other data metric lengths are computed.
y	object of class sf, sfc or sfg, defaults to x
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.

**Details**

Function `dist_fun` should follow the pattern of the distance function `distGeo`: the first two arguments must be 2-column point matrices, the third the semi major axis (radius, in m), the third the ellipsoid flattening.

**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, `areaPolygon` is used for area calculation.

`st_length` returns the length of a `LINestring` or `MULTILINestring` geometry, using the coordinate reference system. `POINT` or `MULTIPOINT` geometries return zero, `POLYGON` or `MULTIPOLYGON` are converted into `LINestring` or `MULTILINestring`, respectively.

If `by_element` is FALSE a dense numeric matrix of dimension `length(x)` by `length(y)`; otherwise a numeric vector of length x or y, the shorter one being recycled.

**See Also**

[st\\_dimension](#)

**Examples**

```
b0 = st_polygon(list(rbind(c(-1,-1), c(1,-1), c(1,1), c(-1,1), c(-1,-1))))
b1 = b0 + 2
b2 = b0 + c(-0.2, 2)
x = st_sfc(b0, b1, b2)
st_area(x)
dist_vincenty = function(p1, p2, a, f) geosphere::distVincentyEllipsoid(p1, p2, a, a * (1-f), f)
line = st_sfc(st_linestring(rbind(c(30,30), c(40,40))), crs = 4326)
st_length(line)
st_length(line, dist_fun = dist_vincenty)
p = st_sfc(st_point(c(0,0)), st_point(c(0,1)), st_point(c(0,2)))
st_distance(p, p)
st_distance(p, p, by_element = TRUE)
```

---

geos\_query

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


---

**Description**

Dimension, simplicity or validity queries on simple feature geometries

**Usage**

```
st_dimension(x, NA_if_empty = TRUE)

st_is_simple(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\_valid returns a logical vector indicating for each geometries of x whether it is valid.

**Examples**

```
x = st_sfc(
  st_point(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)))),
  st_multipoint(),
  st_linestring(),
  st_geometrycollection())
st_dimension(x)
st_dimension(x, FALSE)
```



```

ls = st_linestring(rbind(c(0,0), c(1,1), c(1,0), c(0,1)))
st_is_simple(st_sfc(ls, st_point(c(0,0))))
p1 = st_as_sfc("POLYGON((0 0, 0 10, 10 0, 10 10, 0 0))")
st_is_valid(p1)
st_is_valid(st_sfc(st_point(0:1), p1[[1]]), reason = TRUE)

```

---

geos\_unary

*Geometric unary operations on simple feature geometry sets*


---

### Description

Geometric unary operations on simple feature geometry sets. These are all generics, with methods for sfg, sfc and sf objects, returning an object of the same class.

### Usage

```

st_buffer(x, dist, nQuadSegs = 30)

st_boundary(x)

st_convex_hull(x)

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

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

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

st_polygonize(x)

st_line_merge(x)

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

st_point_on_surface(x)

st_node(x)

st_segmentize(x, dfMaxLength, ...)

```

### Arguments

x	object of class sfg, sfc or sf
dist	numeric; buffer distance for all, or for each of the elements in x; in case 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)

preserveTopology	logical; carry out topology preserving simplification?
dTolerance	numeric; tolerance parameter
bOnlyEdges	logical; if TRUE, return lines, else return polygons
envelope	object of class <code>sfc</code> or <code>sfg</code> 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 <code>st_centroid</code> : 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 <code>x</code> has geographical coordinates (long/lat), <code>dfMaxLength</code> is either a numeric expressed in meter, or an object of class <code>units</code> with length units or unit rad, and segmentation takes place along the great circle, using <a href="#">gcIntermediate</a> .

### Details

`st_triangulate` requires GEOS version 3.4 or above

`st_voronoi` requires GEOS version 3.5 or above

in case of `st_polygonize`, `x` must be an object of class `LINestring` or `MULTILINestring`, or an `sfc` geometry list-column object containing these

in case of `st_line_merge`, `x` must be an object of class `MULTILINestring`, or an `sfc` geometry list-column object containing these

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

`st_node` add nodes to linear geometries at intersections without node

### Value

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

### Examples

```
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"] >= "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, pch=16)
}
```

```

m1s = st_multilinestring(list(matrix(c(0,0,0,1,1,1,0,0),,2,byrow=TRUE)))
st_polygonize(st_sfc(m1s))
m1s = st_multilinestring(list(rbind(c(0,0), c(1,1)), rbind(c(2,0), c(1,1))))
st_line_merge(st_sfc(m1s))
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(1))
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]])

```

---

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 <a href="#">st_drivers</a>

---

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")
```

**Arguments**

drv	character. Name of driver
drivers	data.frame. Table containing driver names and support. Default is from <a href="#">st_drivers</a>
operation	character. What action to check

---

merge.sf	<i>merge method for sf and data.frame object</i>
----------	--

---

**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)
```

---

Ops.sfg	<i>S3 Ops Group Generic Functions (multiply and add/subtract) for affine transformation</i>
---------	---

---

**Description**

Ops functions for simple feature geometry objects (constrained to multiplication and addition)

**Usage**

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

**Arguments**

- e1                    object of class sfg
- e2                    numeric; in case of multiplication an n x n matrix, in case of addition or subtraction a vector of length n, with n the number of dimensions of the geometry

**Value**

object of class sfg

**Examples**

```
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)
plot(w2, axes = TRUE)
```

---

plot

*Plot sf object*


---

**Description**

Plot sf object

blue-pink-yellow color scale

**Usage**

```
## S3 method for class 'sf'
plot(x, y, ..., col = NULL, main, pal = NULL, nbreaks = 10,
     breaks = "pretty", max.plot = if (is.null(n <-
     options("sf_max.plot")[[1]])) 9 else n, key.pos = if (ncol(x) > 2) NULL else
     4, key.size = lcm(1.8))

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

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

```

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

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

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

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

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

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

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

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

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

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

```

### Arguments

x	object of class sf
y	ignored
...	further specifications, see <a href="#">plot_sf</a> and <a href="#">plot</a>
col	color
main	title for plot (NULL to remove)
pal	palette function, similar to <a href="#">rainbow</a> ; if omitted, <a href="#">sf.colors</a> is used
nbreaks	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 <a href="#">classIntervals</a>
max.plot	integer; lower boundary to maximum number of attributes to plot; the default value (9) can be overridden by setting the global option <code>sf_max.plot</code> , e.g. <code>options(sf_max.plot=2)</code>
key.pos	integer; which side to plot a color key: 1 bottom, 2 left, 3 top, 4 right. Set to NULL for no key. Currently ignored if multiple columns are plotted.
key.size	amount of space reserved for the key (labels)
pch	plotting symbol
cex	symbol size
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?
border	color of polygon border
rule	see <a href="#">polypath</a>
xlim	see <a href="#">plot.window</a>
ylim	see <a href="#">plot.window</a>
asp	see below, and see <a href="#">par</a>
axes	logical; should axes be plotted? (default FALSE)
bgc	background color
xaxs	see <a href="#">par</a>
yaxs	see <a href="#">par</a>
lab	see <a href="#">par</a>
setParUsrBB	default FALSE; set the par "usr" bounding box; see below
bgMap	object of class <code>ggmap</code> , or returned by function <code>RgoogleMaps::GetMap</code>
expandBB	numeric; fractional values to expand the bounding box with, in each direction (bottom, left, top, right)
graticule	logical, or object of class <code>crs</code> (e.g., <code>st_crs(4326)</code> for a WGS84 graticule), or object created by <a href="#">st_graticule</a> ; TRUE will give the WGS84 graticule or object returned by <a href="#">st_graticule</a>
col_graticule	color to used for the graticule (if present)
n	integer; number of colors
cutoff.tails	numeric, in [0,0.5] start and end values
alpha	numeric, in [0,1], transparency
categorical	logical; do we want colors for a categorical variable? (see details)

## Details

`plot.sf` maximally plots `max.plot` maps with colors following from attribute columns, one map per attribute. It uses `sf.colors` for default colors. For more control over individual maps, set parameter `mfrow` with `par` prior to plotting, and plot single maps one by one.

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

`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](http://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

```
# plot linestrings:
l1 = st_linestring(matrix(runif(6)-0.5,,2))
l2 = st_linestring(matrix(runif(6)-0.5,,2))
l3 = st_linestring(matrix(runif(6)-0.5,,2))
s = st_sf(a=2:4, b=st_sfc(l1,l2,l3))
plot(s, col = s$a, axes = FALSE)
plot(s, col = s$a)
ll = "+init=epsg:4326 +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"
st_crs(s) = ll
plot(s, col = s$a, axes = TRUE)
plot(s, col = s$a, lty = s$a, lwd = s$a, pch = s$a, type = 'b')
l4 = st_linestring(matrix(runif(6),,2))
plot(st_sf(a=1,b=st_sfc(l4)), add = TRUE)
# plot multilinestrings:
m1 = st_multilinestring(list(l1, l2))
m2 = st_multilinestring(list(l3, l4))
ml = st_sf(a = 2:3, b = st_sfc(m1, m2))
plot(ml, col = ml$a, lty = ml$a, lwd = ml$a, pch = ml$a, type = 'b')
# plot points:
p1 = st_point(c(1,2))
p2 = st_point(c(3,3))
p3 = st_point(c(3,0))
p = st_sf(a=2:4, b=st_sfc(p1,p2,p3))
plot(p, col = s$a, axes = TRUE)
plot(p, col = s$a)
plot(p, col = p$a, pch = p$a, cex = p$a, bg = s$a, lwd = 2, lty = 2, type = 'b')
```



```

p4 = st_point(c(2,2))
plot(st_sf(a=1, st_sfc(p4)), add = TRUE)
# multipoints:
mp1 = st_multipoint(matrix(1:4,2))
mp2 = st_multipoint(matrix(5:8,2))
mp = st_sf(a = 2:3, b = st_sfc(mp1, mp2))
plot(mp)
plot(mp, col = mp$a, pch = mp$a, cex = mp$a, bg = mp$a, lwd = mp$a, lty = mp$a, type = 'b')
# polygon:
outer = matrix(c(0,0,10,0,10,10,0,10,0,0),ncol=2, byrow=TRUE)
hole1 = matrix(c(1,1,1,2,2,2,2,1,1,1),ncol=2, byrow=TRUE)
hole2 = matrix(c(5,5,5,6,6,6,6,5,5,5),ncol=2, byrow=TRUE)
p11 = st_polygon(list(outer, hole1, hole2))
p12 = st_polygon(list(outer+10, hole1+10, hole2+10))
po = st_sf(a = 2:3, st_sfc(p11,p12))
plot(po, col = po$a, border = rev(po$a), lwd=3)
# multipolygon
r10 = matrix(rep(c(0,10),each=5),5)
p11 = list(outer, hole1, hole2)
p12 = list(outer+10, hole1+10, hole2+10)
p13 = list(outer+r10, hole1+r10, hole2+r10)
mpo1 = st_multipolygon(list(p11,p12))
mpo2 = st_multipolygon(list(p13))
mpo = st_sf(a=2:3, b=st_sfc(mpo1,mpo2))
plot(mpo, col = mpo$a, border = rev(mpo$a), lwd = 2)
# geometrycollection:
gc1 = st_geometrycollection(list(mpo1, st_point(c(21,21)), 11 * 2 + 21))
gc2 = st_geometrycollection(list(mpo2, 12 - 2, 13 - 2, st_point(c(-1,-1))))
gc = st_sf(a=2:3, b = st_sfc(gc1,gc2))
plot(gc, cex = gc$a, col = gc$a, border = rev(gc$a) + 2, lwd = 2)
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	<i>Convert raw vector(s) into hexadecimal character string(s)</i>
----------	---

---

**Description**

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

**Usage**

```
rawToHex(x)
```

**Arguments**

x	raw vector, or list with raw vectors
---	--------------------------------------

---

sf	<i>Create sf object</i>
----	-------------------------

---

**Description**

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

**Usage**

```
st_sf(..., agr = NA_agr_, row.names,
      stringsAsFactors = default.stringsAsFactors(), crs, precision,
      sf_column_name = NULL)
```

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

**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 <a href="#">st_as_sfc</a> .
agr	character vector; see details below.
row.names	row.names for the created sf object
stringsAsFactors	logical; logical: should character vectors be converted to factors? The ‘factory-fresh’ default is TRUE, but this can be changed by setting <code>options(stringsAsFactors = FALSE)</code> .
crs	coordinate reference system: integer with the EPSG code, or character with proj4string
precision	numeric; see <a href="#">st_as_binary</a>

<code>sf_column_name</code>	character; name of the active list-column with simple feature geometries; in case there are more than one and <code>sf_column_name</code> is not given, the first one is taken.
<code>x</code>	object of class <code>sf</code>
<code>i</code>	record selection, see <a href="#">[.data.frame]</a>
<code>j</code>	variable selection, see <a href="#">[.data.frame]</a>
<code>drop</code>	logical, default FALSE; if TRUE drop the geometry column and return a <code>data.frame</code> , else make the geometry sticky and return a <code>sf</code> object.
<code>op</code>	function; geometrical binary predicate function to apply when <code>i</code> is a simple feature object

## 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

```
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,]
```

---

sfc *Create simple feature geometry list column*

---

### Description

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

### Usage

```
st_sfc(..., crs = NA_crs_, precision = 0)
```

### Arguments

...	zero or more simple feature geometries (objects of class <code>sfg</code> ), 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 <a href="#">st_as_binary</a>

### 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()
```

---

 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)
```

**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 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**

```
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, ..., digits = 0)

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

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

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

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

```

### Arguments

<code>x</code>	for <code>st_point</code> , numeric vector (or one-row-matrix) of length 2, 3 or 4; for <code>st_linestring</code> and <code>st_multipoint</code> , numeric matrix with points in rows; for <code>st_polygon</code> and <code>st_multilinestring</code> , list with numeric matrices with points in rows; for <code>st_multipolygon</code> , list of lists with numeric matrices; for <code>st_geometrycollection</code> list with (non-geometrycollection) simple feature objects
<code>dim</code>	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
<code>dims</code>	character; specify dimensionality in case of an empty (NULL) geometrycollection, in which case <code>x</code> is the empty <code>list()</code> .
<code>...</code>	objects to be pasted together into a single simple feature
<code>digits</code>	integer; number of characters to be printed (max 30; 0 means print everything)
<code>n</code>	integer; number of elements to be selected
<code>recursive</code>	logical; ignored
<code>flatten</code>	logical; if TRUE, try to simplify results; if FALSE, return geometrycollection containing all objects

### Details

"XYZ" refers to coordinates where the third dimension represents altitude, "XYM" refers to three-dimensional coordinates where the third dimension refers to something else ("M" for measure); checking of the sanity of `x` may be only partial.

When `flatten=TRUE`, this method may merge points into a multipoint structure, and may not preserve order, and hence cannot be reverted. When given fish, it returns fish soup.

**Value**

object of the same nature as `x`, but with appropriate class attribute set

`as.matrix` returns the set of points that form a geometry as a single matrix, where each point is a row; use `unlist(x, recursive = FALSE)` to get sets of matrices.

**Examples**

```
(p1 = st_point(c(1,2)))
class(p1)
st_bbox(p1)
(p2 = st_point(c(1,2,3)))
class(p2)
(p3 = st_point(c(1,2,3), "XYM"))
pts = matrix(1:10, , 2)
(mp1 = st_multipoint(pts))
pts = matrix(1:15, , 3)
(mp2 = st_multipoint(pts))
(mp3 = st_multipoint(pts, "XYM"))
pts = matrix(1:20, , 4)
(mp4 = st_multipoint(pts))
pts = matrix(1:10, , 2)
(ls1 = st_linestring(pts))
pts = matrix(1:15, , 3)
(ls2 = st_linestring(pts))
(ls3 = st_linestring(pts, "XYM"))
pts = matrix(1:20, , 4)
(ls4 = st_linestring(pts))
outer = matrix(c(0,0,10,0,10,10,0,10,0,0),ncol=2, byrow=TRUE)
hole1 = matrix(c(1,1,1,2,2,2,2,1,1,1),ncol=2, byrow=TRUE)
hole2 = matrix(c(5,5,5,6,6,6,6,5,5,5),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(0,0,10,0,10,10,0,10,0,0),ncol=2, byrow=TRUE)
hole1 = matrix(c(1,1,1,2,2,2,2,1,1,1),ncol=2, byrow=TRUE)
hole2 = matrix(c(5,5,5,6,6,6,6,5,5,5),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(p1, 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_geometrycollection(list(st_point(1:2), st_linestring(matrix(1:6,3))),
  st_geometrycollection(list(st_multilinestring(list(matrix(11:16,3))))))
c(st_geometrycollection(list(st_point(1:2), st_linestring(matrix(1:6,3))),
  st_multilinestring(list(matrix(11:16,3))), st_point(5:6),
  st_geometrycollection(list(st_point(10:11))))

```

---

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

```
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	<i>Convert sfc object to an WKB object</i>
--------------	--

---

**Description**

Convert sfc object to an WKB object

**Usage**

```
st_as_binary(x, ...)

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

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

**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>. 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**

```
x = st_sfc(st_point(c(1/3, 1/6)), precision = 1000)
st_as_sfc(st_as_binary(x)) # rounds
```

---

st_as_grob	<i>Convert sf* object to a grob</i>
------------	-------------------------------------

---

**Description**

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

**Usage**

```
st_as_grob(x, ..., units = "native")
```

**Arguments**

x	object to be converted into an object class grob
...	passed on to the xxxGrob function, e.g. gp = gpar(col = 'red')
units	units; see <a href="#">unit</a>

---

st_as_sf	<i>Convert foreign object to an sf object</i>
----------	---

---

**Description**

Convert foreign object to an sf object

**Usage**

```
st_as_sf(x, ...)

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

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

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

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

```
## S3 method for class 'ppp'
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 <a href="#">st_sf</a> , might included named arguments crs or precision
agr	character vector; see details section of <a href="#">st_sf</a>
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 <a href="#">st_point</a> (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

### 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(0 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
m = matrix(c(3, 0), 5, 2, byrow = TRUE)
z = x + m
z1 = x1 + m
z2 = x2 + m
z3 = x3 + m
p1 = Polygons(list( Polygon(x[5:1,]), Polygon(x2), Polygon(x3),
  Polygon(y[5:1,]), Polygon(y1), Polygon(x1), Polygon(y3)), "ID1")
p2 = Polygons(list( Polygon(z[5:1,]), Polygon(z2), Polygon(z3), Polygon(z1)),
  "ID2")
if (require("rgeos")) {
  r = createSPComment(SpatialPolygons(list(p1,p2)))
  comment(r)
  comment(r@polygons[[1]])
  scan(text = comment(r@polygons[[1]]), quiet = TRUE)
  library(sf)
  a = st_as_sf(r)
  summary(a)
}
demo(meuse, ask = FALSE, echo = FALSE)
summary(st_as_sf(meuse))
summary(st_as_sf(meuse.grid))
summary(st_as_sf(meuse.area))
summary(st_as_sf(meuse.riv))
summary(st_as_sf(as(meuse.riv, "SpatialLines")))
pol.grd = as(meuse.grid, "SpatialPolygonsDataFrame")
summary(st_as_sf(pol.grd))
summary(st_as_sf(as(pol.grd, "SpatialLinesDataFrame")))
## Not run:
  require(spatstat)
  g = st_as_sf(gorillas)
  # select only the points:
  g[st_is(g, "POINT"),]

## End(Not run)
## Not run: # because of spatstat interfering with units
if (require(spatstat)) {
  data(chicago)
  plot(st_as_sf(chicago["label"]))
  plot(st_as_sf(chicago[-1,"label"]))
}

## End(Not run)

```

---

st\_as\_sfc

---

*Convert foreign geometry object to an sfc object*


---

### Description

Convert foreign geometry object to an sfc object

**Usage**

```

## 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 'character'
st_as_sfc(x, crs = NA_integer_, ...)

## 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 'SpatialPixels'
st_as_sfc(x, ..., precision = 0)

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

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

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

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

```

**Arguments**

x	object to convert
...	further arguments
crs	integer or character; coordinate reference system for the geometry, see <a href="#">st_crs()</a>
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 <a href="https://www.gaia-gis.it/gaia-sins/BLOB-Geometry.html">https://www.gaia-gis.it/gaia-sins/BLOB-Geometry.html</a> ; 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).
precision	precision value; see <a href="#">st_as_binary</a>
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 the [well-known-text](#) or [Postgis EWKT](#) representations of a single geometry for each vector element.

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

### Examples

```
wkb = structure(list("01010000204071000000000000801A064100000000AC5C1441"), class = "WKB")
st_as_sfc(wkb, EWKB = TRUE)
wkb = structure(list("0x01010000204071000000000000801A064100000000AC5C1441"), class = "WKB")
st_as_sfc(wkb, EWKB = TRUE)
st_as_sfc("SRID=3978;LINESTRING(1663106 -105415,1664320 -104617)")
```

---

st_as_text	<i>Return Well-known Text representation of simple feature geometry or coordinate reference system</i>
------------	--

---

### Description

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

### Usage

```
## 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
...	passed on to WKT_name
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**

To suppress printing of SRID, EWKT=FALSE can be passed as parameter.

**Examples**

```
st_as_text(st_point(1:2))
```

---

st_bbox	<i>Return bounding of a simple feature or simple feature set</i>
---------	--

---

**Description**

Return bounding of a simple feature or simple feature set

**Usage**

```
NA_bbox_
```

```
st_bbox(obj)
```

```
\method{st_bbox}{POINT}(obj)
```

```
\method{st_bbox}{MULTIPOINT}(obj)
```

```
\method{st_bbox}{LINESTRING}(obj)
```

```
\method{st_bbox}{POLYGON}(obj)
```

```
\method{st_bbox}{MULTILINESTRING}(obj)
```

```
\method{st_bbox}{MULTIPOLYGON}(obj)
```

```
\method{st_bbox}{GEOMETRYCOLLECTION}(obj)
```

```
\method{st_bbox}{MULTISURFACE}(obj)
```

```
\method{st_bbox}{MULTICURVE}(obj)
```

```
\method{st_bbox}{CURVEPOLYGON}(obj)
\method{st_bbox}{COMPOUNDCURVE}(obj)
\method{st_bbox}{POLYHEDRALSURFACE}(obj)
\method{st_bbox}{TIN}(obj)
\method{st_bbox}{TRIANGLE}(obj)
\method{st_bbox}{CIRCULARSTRING}(obj)
\method{st_bbox}{sfc}(obj)
\method{st_bbox}{sf}(obj)
```

### Arguments

obj                    object to compute the bounding box from

### Format

An object of class bbox of length 4.

### Details

NA\_bbox\_ is the bbox object with a missing value.

### 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

```
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_cast	<i>Cast geometry to another type: either simplify, or cast explicitly</i>
---------	---

---

**Description**

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

**Usage**

```
## 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("MULTIPOLYGON", "MULTILINESTRING", "MULTIPOINT", "POLYGON", "LINESTRING", "POINT"),
         function(x) st_cast(xg, x))
}
```

```

st_sfc(cast_all(mpl))
## no closing coordinates should remain for multipoint
any(duplicated(unclass(st_cast(mpl, "MULTIPOINT")))) ## should be FALSE
## number of duplicated coordinates in the linestrings should equal the number of polygon rings
## (... in this case, won't always be true)
sum(duplicated(do.call(rbind, unclass(st_cast(mpl, "MULTILINESTRING"))))
    ) == sum(unlist(lapply(mpl, length))) ## should be TRUE

p1 <- structure(c(0, 1, 3, 2, 1, 0, 0, 0, 2, 4, 4, 0), .Dim = c(6L, 2L))
p2 <- structure(c(1, 1, 2, 1, 1, 2, 2, 1), .Dim = c(4L, 2L))
st_polygon(list(p1, p2))
m1s <- st_cast(nc$geometry[[4]], "MULTILINESTRING")
st_sfc(cast_all(m1s))
mpt <- st_cast(nc$geometry[[4]], "MULTIPOINT")
st_sfc(cast_all(mpt))
p1 <- st_cast(nc$geometry[[4]], "POLYGON")
st_sfc(cast_all(p1))
ls <- st_cast(nc$geometry[[4]], "LINESTRING")
st_sfc(cast_all(ls))
pt <- st_cast(nc$geometry[[4]], "POINT")
## st_sfc(cast_all(pt)) ## Error: cannot create MULTIPOLYGON from POINT
st_sfc(lapply(c("POINT", "MULTIPOINT"), function(x) st_cast(pt, x)))
s = st_multipoint(rbind(c(1,0)))
st_cast(s, "POINT")

```

---

st\_cast\_sfc\_default    *Coerce geometry to MULTI\* geometry*

---

### Description

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

### Usage

```
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

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

## S3 method for class 'sfg'
st_collection_extract(x, type = c("POLYGON", "POINT",
  "LINESTRING"), warn = FALSE)

## S3 method for class 'sfc'
st_collection_extract(x, type = c("POLYGON", "POINT",
  "LINESTRING"), warn = FALSE)

## S3 method for class 'sf'
st_collection_extract(x, type = c("POLYGON", "POINT",
  "LINESTRING"), warn = FALSE)
```

## Arguments

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

## Value

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

## Examples

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

```

multipoly <- st_multipolygon(list(poly1, poly2))

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

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

## A GEOMETRYCOLLECTION
aa <- rbind(st_sf(a=1, geom = st_sfc(i)),
st_sf(a=2, geom = st_sfc(j)))

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

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

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

st_collection_extract(bb, "POLYGON")

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

st_collection_extract(cc, "POLYGON")

```

---

st_coordinates	<i>retrieve coordinates in matrix form</i>
----------------	--

---

### Description

retrieve coordinates in matrix form

### Usage

```
st_coordinates(x, ...)
```

**Arguments**

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

**Value**

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

---

st_crs	<i>Retrieve coordinate reference system from object</i>
--------	---

---

**Description**

Retrieve coordinate reference system from sf or sfc object  
 Set or replace retrieve coordinate reference system from object

**Usage**

```
st_crs(x, ...)  
  
## S3 method for class 'sf'  
st_crs(x, ...)  
  
## S3 method for class 'numeric'  
st_crs(x, ...)  
  
## S3 method for class 'character'  
st_crs(x, ..., wkt)  
  
## S3 method for class 'sfc'  
st_crs(x, ...)  
  
## S3 method for class 'bbox'  
st_crs(x, ...)  
  
## S3 method for class 'crs'  
st_crs(x, ...)  
  
st_crs(x) <- value  
  
## S3 replacement method for class 'sf'  
st_crs(x) <- value
```

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

st_set_crs(x, value)

NA_crs_

## S3 method for class 'crs'
is.na(x)

## S3 method for class 'crs'
x$name
```

### Arguments

x	numeric, character, or object of class <code>sf</code> or <code>sfc</code>
...	ignored
wkt	character well-known-text representation of the crs
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**

```
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=km")$b      # numeric
st_crs("+init=epsg:3857 +units=km")$units # character
```

---

st\_drivers

*Get GDAL drivers*


---

**Description**

Get a list of the available GDAL drivers

**Usage**

```
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.



**Value**

a `data.frame` with driver metadata

**Examples**

```
st_drivers()
```

---

st_geohash	<i>compute geohash from (average) coordinates (requires lwgeom)</i>
------------	---

---

**Description**

compute geohash from (average) coordinates (requires lwgeom)

**Usage**

```
st_geohash(x, precision = 0)
```

**Arguments**

x	object of class <code>sf</code> , <code>sfc</code> or <code>sfg</code>
precision	integer; precision (length) of geohash returned; when omitted, precision 10 is taken.

**Details**

see <http://geohash.org/> or <https://en.wikipedia.org/wiki/Geohash>. in case a geometry contains more than one point, the geohash for the average of the points in the geometry is returned.

**Value**

character vector with geohashes

**Examples**

```
if (!is.na(sf_extSoftVersion()["lwgeom"])) {
  st_geohash(st_sfc(st_point(c(1.5,3.5)), st_point(c(0,90))), 2)
  st_geohash(st_sfc(st_point(c(1.5,3.5)), st_point(c(0,90))), 10)
}
```

---

st\_geometry

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


---

## Description

Get, set, or replace geometry from an sf object

## Usage

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

st_geometry(obj, ...)

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

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

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

st_geometry(x) <- value

st_set_geometry(x, value)
```

## 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.

**Value**

st\_geometry returns an object of class `sfc`, a list-column with geometries

st\_geometry returns an object of class `sf`. Assigning geometry to a `data.frame` creates an `sf` object, assigning it to an `sf` object replaces the geometry list-column.

**Examples**

```
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	<i>Return geometry type of an object</i>
------------------	--

---

**Description**

Return geometry type of an object, as a factor

**Usage**

```
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 <i>sf</i> , <i>sfc</i> or <i>sfg</i> or numeric vector with bounding box given as (minx, miny, maxx, maxy).
crs	object of class <i>crs</i> , with the display coordinate reference system
datum	either an object of class <i>crs</i> with the coordinate reference system for the graticules, or <i>NULL</i> in which case a grid in the coordinate system of <i>x</i> is drawn, or <i>NA</i> , in which case an empty <i>sf</i> 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**

```

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_crs(4326), axes = TRUE, lon = seq(-60,-130,by=-10))

```

---

st\_interpolate\_aw

*Areal-weighted interpolation of polygon data*


---

**Description**

Areal-weighted interpolation of polygon data

**Usage**

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

**Arguments**

x	object of class sf, for which we want to aggregate attributes
to	object of class sf or sfc, with the target geometries
extensive	logical; if TRUE, the attribute variables are assumed to be spatially extensive (like population) and the sum is preserved, otherwise, spatially intensive (like population density) and the mean is preserved.

**Examples**

```
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)
sum(a2$BIR74) / sum(nc$BIR74)
a1$intensive = a1$BIR74
a1$extensive = a2$BIR74
plot(a1[c("intensive", "extensive")])
```

---

st\_is

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


---

**Description**

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

**Usage**

```
st_is(x, type)
```

**Arguments**

x	object of class sf, sfc or sfg
type	character; class, or set of classes, to test against

**Examples**

```
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"))
```

---

st_is_longlat	<i>Assert whether simple feature coordinates are longlat degrees</i>
---------------	--

---

**Description**

Assert whether simple feature coordinates are longlat degrees

**Usage**

```
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	<i>jitter geometries</i>
-----------	--------------------------

---

**Description**

jitter geometries

**Usage**

```
st_jitter(x, amount, factor = 0.002)
```

**Arguments**

x                    object of class [sf](#) or [sfc](#)

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

factor               numeric; fractional amount of jittering to be applied

**Details**

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

**Examples**

```
nc = 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_geometry(nc))
plot(st_jitter(st_geometry(nc), factor = .01), add = TRUE, col = '#ff8888')
```

---

st_join	<i>spatial left or inner join</i>
---------	-----------------------------------

---

**Description**

spatial left or inner join

**Usage**

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

**Arguments**

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

**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**

```

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))

```

---

**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**

```
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 <code>round(density * st_length(geom))</code> .
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 overrides n, density and type.

**Examples**

```
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))
```

---

st_make_grid	<i>Make a rectangular grid over the bounding box of a sf or sfc object</i>
--------------	--

---

**Description**

Make a rectangular grid over the bounding box of a sf or sfc object

**Usage**

```
st_make_grid(x, cellsize = c(diff(st_bbox(x))[c(1, 3)]), diff(st_bbox(x))[c(2,
4)])/n, offset = st_bbox(x)[1:2], n = c(10, 10), crs = if (missing(x))
NA_crs_ else st_crs(x), what = "polygons")
```

**Arguments**

x	object of class sf or sfc
cellsize	target cellsize
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
what	character; one of: "polygons", "corners", or "centers"

**Value**

Object of class `sfc` (simple feature geometry list column) with, depending on `what`, rectangular polygons, corner points of these polygons, or center points of these polygons.

**Examples**

```
plot(st_make_grid(what = "centers"), axes = TRUE)
plot(st_make_grid(what = "corners"), add = TRUE, col = 'green', pch=3)
```

---

st_precision	<i>Get precision</i>
--------------	----------------------

---

**Description**

Get precision

Set precision

**Usage**

```
st_precision(x)

st_set_precision(x, precision)

st_precision(x) <- value
```

**Arguments**

<code>x</code>	object of class <code>sfc</code> or <code>sf</code>
<code>precision</code>	numeric; see <a href="#">st_as_binary</a> for how to do this.
<code>value</code>	precision value

**Details**

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\\_as\\_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](#), [write\\_sf](#) or [st\\_write\\_db](#), and (if present) for `liblwgeom` ([st\\_make\\_valid](#)); 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.

**Examples**

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

---

<code>st_read</code>	<i>Read simple features or layers from file or database</i>
----------------------	---

---

**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 binary

**Usage**

```
st_read(dsn, layer, ..., options = NULL, quiet = FALSE,
        geometry_column = 1L, type = 0, promote_to_multi = TRUE,
        stringsAsFactors = default.stringsAsFactors(), int64_as_string = FALSE)

read_sf(..., quiet = TRUE, stringsAsFactors = FALSE)

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

**Arguments**

<code>dsn</code>	data source name (interpretation varies by driver - for some drivers, <code>dsn</code> is a file name, but may also be a folder, or contain the name and access credentials of a database); in case of GeoJSON, <code>dsn</code> may be the character string holding the geojson data
<code>layer</code>	layer name (varies by driver, may be a file name without extension); in case layer is missing, <code>st_read</code> will read the first layer of <code>dsn</code> , give a warning and (unless <code>quiet = TRUE</code> ) print a message when there are multiple layers, or give an error if there are no layers in <code>dsn</code> .
<code>...</code>	parameter(s) passed on to <a href="#">st_as_sf</a>
<code>options</code>	character; driver dependent dataset open options, multiple options supported.
<code>quiet</code>	logical; suppress info on name, driver, size and spatial reference, or signaling no or multiple layers
<code>geometry_column</code>	integer or character; in case of multiple geometry fields, which one to take?
<code>type</code>	integer; ISO number of desired simple feature type; see details. If left zero, and <code>promote_to_multi</code> is <code>TRUE</code> , 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 Multi-LineString, 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 <sup>53</sup> ).
conn	open database connection
table	table name
query	SQL query to select records; see details
geom_column	character or integer: indicator of name or position of the geometry column; if not provided, the last column of type character is chosen
EWKB	logical; is the WKB is of type EWKB? if missing, defaults to TRUE if conn is of class codePostgreSQLConnection or PqConnection, and to FALSE otherwise

## 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 type values 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>.

`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. `write_sf` delete layers by default: it overwrites existing files.

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)

in case `geom_column` is missing: if `table` is missing, this function will try to read the name of the geometry column from `table` `geometry_columns`, in other cases, or when this fails, the `geom_column` is assumed to be the last column of mode character. If `table` is missing, the SRID cannot be read and resolved into a proj4string by the database, and a warning will be given.

## 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**

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

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

## End(Not run)
# read geojson from string:
geojson_txt <- paste("{\"type\":\"MultiPoint\",\"coordinates\":",
  "[[[3.2,4],[3,4.6],[3.8,4.4],[3.5,3.8],[3.4,3.6],[3.9,4.5]]]"}")
x = read_sf(geojson_txt)
x
## Not run:
library(RPostgreSQL)
try(conn <- dbConnect(PostgreSQL(), dbname = "postgis"))
if (exists("conn") && !inherits(conn, "try-error")) {
  x = st_read_db(conn, "meuse", query = "select * from meuse limit 3;")
  x = st_read_db(conn, table = "public.meuse")
  print(st_crs(x)) # SRID resolved by the database, not by GDAL!
  dbDisconnect(conn)
}

## End(Not run)
```

---

st\_relate

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

---

**Description**

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

**Usage**

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

**Arguments**

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

**Value**

In case pattern is not given, st\_relate returns a dense character matrix; element [i,j] has nine characters, referring to the DE9-IM relationship between x[i] and y[j], encoded as IxIy,IxBx,IxEy,BxIy,BxBx,BxEy,ExIy,ExBx where I refers to interior, B to boundary, and E to exterior, and e.g. BxIy the dimensionality of the intersection of the 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**

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

---

st\_sample

*sample points on or in (sets of) spatial features*


---

**Description**

sample points on or in (sets of) spatial features

**Usage**

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

**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 <a href="#">sample</a> for multipoint sampling
type	character; indicates the spatial sampling type; only random is implemented right now

**Details**

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>

**Examples**

```
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)
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)
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(90,90),c(0,90),c(0,0)))))) # NOT long/lat:
plot(x)
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))
p = st_sample(x, 1000)
pt = st_multipoint(matrix(1:20, ,2))
st_sample(p, 3)
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(2,2.00001))))
st_sample(ls, 80)
```

---

st\_split

*Return a collection of geometries resulting by splitting a geometry*


---

**Description**

Return a collection of geometries resulting by splitting a geometry



**Usage**

```
st_split(x, y)
```

**Arguments**

**x** object with geometries to be splitted

**y** object split with (blade); if y contains more than one feature geometry, the geometries are [st\\_combined](#)

**Details**

st\_split is only available if the package was linked against liblwgeom, which is currently not the case for the binary CRAN distributions; see the package source code repository for instructions how to install liblwgeom. The example below shows how to run-time check the availability of liblwgeom.

**Value**

object of the same class as x

**Examples**

```
l = st_as_sfc('MULTILINESTRING((10 10, 190 190), (15 15, 30 30, 100 90))')
pt = st_sfc(st_point(c(30,30)))
if (!is.na(sf_extSoftVersion()["lwgeom"])) {
  st_split(l, pt)
}
```

---

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")

st_wrap_dateline(x, options = "WRAPDATELINE=YES", quiet = TRUE)

```

### Arguments

x	object of class sf, sfc or sfg
crs	coordinate reference system: integer with the EPSG code, or character with proj4string
...	ignored
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; if FALSE, arguments check and partial get ignored, and projection is directly carried out by lwgeom_transform which ignores the GDAL api. This allows proj.4 parameters such as +over to act ( <a href="https://github.com/r-spatial/sf/issues/511">https://github.com/r-spatial/sf/issues/511</a> ) and certain projections such as wintri which do not have an inverse, to work ( <a href="https://github.com/r-spatial/sf/issues/509">https://github.com/r-spatial/sf/issues/509</a> ). This requires that sf is built by linking to liblwgeom.
type	character; one of proj, ellps, datum or units
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.

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

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

### Examples

```

p1 = st_point(c(7,52))
p2 = st_point(c(-30,20))
sfc = st_sfc(p1, p2, crs = 4326)
sfc
st_transform(sfc, 3857)
st_transform(st_sf(a=2:1, geom=sfc), "+init=epsg:3857")
nc = st_read(system.file("shape/nc.shp", package="sf"))

```

```

st_area(nc[1,]) # area, using geosphere::areaPolygon
st_area(st_transform(nc[1,], 32119)) # NC state plane, m
st_area(st_transform(nc[1,], 2264)) # NC state plane, US foot
library(units)
set_units(st_area(st_transform(nc[1,], 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))

```

---

st\_viewport

*Create viewport from sf, sfc or sfg object*


---

### Description

Create viewport from sf, sfc or sfg object

### Usage

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

### Arguments

x	object of class sf, sfc or sfg object
...	parameters passed on to <a href="#">viewport</a>
bbox	the bounding box used for aspect ratio
asp	numeric; target aspect ratio (y/x), see <a href="#">Details</a>

### 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

```

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

```

---

 st\_write

 Write simple features object to file or database
 

---

### Description

Write simple features object to file or database

Write simple feature table to a spatial database

### Usage

```
st_write(obj, dsn, layer = file_path_sans_ext(basename(dsn)),
  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)
```

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

```
st_write_db(conn = NULL, obj, table = deparse(substitute(obj)),
  geom_name = "wkb_geometry", ..., drop = FALSE, debug = FALSE,
  binary = TRUE, append = FALSE)
```

### 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)
layer	layer name (varies by driver, may be a file name without extension); if layer is missing, the <code>basename</code> of dsn is taken.
driver	character; driver name to be used, if missing, a driver name is guessed from dsn; <code>st_drivers()</code> returns the drivers that are available with their properties; links to full driver documentation are found at <a href="http://www.gdal.org/ogr_formats.html">http://www.gdal.org/ogr_formats.html</a> .
...	ignored for <code>st_write</code> , for <code>st_write_db</code> arguments passed on to <code>dbWriteTable</code>
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 <code>as.numeric</code> .
update	logical; FALSE by default for single-layer drivers but TRUE by default for database drivers as defined by <code>db_drivers</code> . 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)
conn	open database connection
table	character; name for the table in the database, possibly of length 2, c("schema", "name"); default schema is public
geom_name	name of the geometry column in the database
drop	logical; should table be dropped first?
debug	logical; print SQL statements to screen before executing them.
binary	logical; use well-known-binary for transfer?
append	logical; append to table? (NOTE: experimental, might not work)

### 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.

`st_write_db` was written with help of Josh London, see <https://github.com/r-spatial/sf/issues/285>

### See Also

[st\\_drivers](#)

### Examples

```
nc = st_read(system.file("shape/nc.shp", package="sf"))
st_write(nc, "nc.shp")
st_write(nc, "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)
st_write(meuse_sf, "meuse.csv", layer_options = "GEOMETRY=AS_XY") # writes X and Y as columns
st_write(meuse_sf, "meuse.csv", layer_options = "GEOMETRY=AS_WKT", delete_dsn=TRUE) # overwrites
## Not run:
library(sp)
example(meuse, ask = FALSE, echo = FALSE)
try(st_write(st_as_sf(meuse), "PG:dbname=postgis", "meuse_sf",
  layer_options = c("OVERWRITE=yes", "LAUNDER=true")))
demo(nc, ask = FALSE)
try(st_write(nc, "PG:dbname=postgis", "sids", layer_options = "OVERWRITE=true"))

## End(Not run)
## Not run:
library(sp)
data(meuse)
sf = st_as_sf(meuse, coords = c("x", "y"), crs = 28992)
library(RPostgreSQL)
try(conn <- dbConnect(PostgreSQL(), dbname = "postgis"))
if (exists("conn") && !inherits(conn, "try-error"))
```

```

st_write_db(conn, sf, "meuse_tbl", drop = FALSE)

## End(Not run)

```

---

st\_zm

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


---

### Description

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

### Usage

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

### Arguments

x	object of class sfg, sfc or sf
...	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

```

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	<i>Summarize simple feature column</i>
-------------	--

---

**Description**

Summarize simple feature column

**Usage**

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

**Arguments**

object	object of class sfc
...	ignored
maxsum	maximum number of classes to summarize the simple feature column to
maxp4s	maximum number of characters to print from the PROJ.4 string

---

tibble	<i>Summarize simple feature type for tibble</i>
--------	---

---

**Description**

Summarize simple feature type for tibble

Summarize simple feature item for tibble

**Usage**

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

```
obj_sum.sfc(x)
```

**Arguments**

x	object of class sfc
...	ignored

---

valid	<i>Make an invalid geometry valid</i>
-------	---------------------------------------

---

**Description**

Make an invalid geometry valid

**Usage**

```
st_make_valid(x)
```

**Arguments**

x                    object of class sfg, SFG or SF

**Details**

`st_make_valid` uses the `lwgeom_makevalid` method also used by the PostGIS command `ST_makevalid`. It is only available if the package was linked against `liblwgeom`, which is currently not the case for the binary CRAN distributions; see the package source code repository for instructions how to install `liblwgeom`. The example below shows how to run-time check the availability of `liblwgeom`.

**Value**

Object of the same class as x

**Examples**

```
x = st_sfc(st_polygon(list(rbind(c(0,0),c(0.5,0),c(0.5,0.5),c(0.5,0),c(1,0),c(1,1),c(0,1),c(0,0)))))
if (!is.na(sf_extSoftVersion()["lwgeom"])) {
  suppressWarnings(st_is_valid(x))
  y = st_make_valid(x)
  st_is_valid(y)
  y %>% st_cast()
}
```



# Index

## \*Topic **datasets**

- db\_drivers, 6
- extension\_map, 10
- prefix\_map, 25
- st\_agr, 32
- st\_bbox, 39
- st\_crs, 46

## \*Topic **data**

- bgMap, 5
- [.data.frame, 27
- [.sf (sf), 26
- \$.crs (st\_crs), 46

- aggregate, 4, 42, 59
- aggregate (aggregate.sf), 3
- aggregate.sf, 3
- anti\_join.sf (dplyr), 7
- areaPolygon, 15
- arrange.sf (dplyr), 7
- as, 4
- as.data.frame, 7
- as.matrix.sfg (st), 29
- as\_Spatial (as), 4

- basename, 68
- bgMap, 5
- bind, 5
- bind\_cols, 6
- bpy.colors, 24

- c, 14
- c.sfg, 14
- c.sfg (st), 29
- cbind, 6
- cbind.sf (bind), 5
- classIntervals, 23
- coerce, sf, Spatial-method (as), 4
- coerce, sfc, Spatial-method (as), 4
- coerce, Spatial, sf-method (as), 4
- coerce, Spatial, sfc-method (as), 4

- data.frame, 6
- db\_drivers, 6
- distGeo, 15
- distinct.sf (dplyr), 7
- dplyr, 7

- extension\_map, 10

- filter.sf (dplyr), 7
- format.sfg (st), 29
- full\_join.sf (dplyr), 7

- g (bgMap), 5
- gather.sf (dplyr), 7
- gcIntermediate, 18
- geos\_binary\_ops, 10
- geos\_binary\_pred, 11
- geos\_combine, 13
- geos\_measures, 14
- geos\_query, 16
- geos\_unary, 17
- getwd, 62
- group\_by.sf (dplyr), 7
- gUnaryUnion, 14
- gUnion, 14

- head.sfg (st), 29

- inner\_join.sf (dplyr), 7
- intersect, 11
- is.na.crs (st\_crs), 46
- is\_driver\_available, 19
- is\_driver\_can, 19

- left\_join, 8, 9, 56
- left\_join.sf (dplyr), 7

- merge, 56
- merge.sf, 20
- mutate.sf (dplyr), 7

NA\_agr\_ (st\_agr), 32  
 NA\_bbox\_ (st\_bbox), 39  
 NA\_crs\_ (st\_crs), 46  
 nest, 9  
 nest.sf (dplyr), 7  
  
 obj\_sum.sfc (tibble), 71  
 Ops.sfg, 20  
  
 par, 23  
 plot, 21, 22  
 plot.window, 23  
 plot\_sf, 22  
 plot\_sf (plot), 21  
 polypath, 23  
 prefix\_map, 25  
 print.sfg (st), 29  
 print.sgbp (sgbp), 29  
  
 rainbow, 22  
 rawToHex, 26  
 rbind, 6  
 rbind.sf (bind), 5  
 read\_sf (st\_read), 60  
 rename.sf (dplyr), 7  
 right\_join.sf (dplyr), 7  
  
 sample, 64  
 sample\_frac.sf (dplyr), 7  
 sample\_n.sf (dplyr), 7  
 select.sf (dplyr), 7  
 semi\_join.sf (dplyr), 7  
 separate, 9  
 separate.sf (dplyr), 7  
 set\_units, 15  
 setdiff, 11  
 sf, 4, 8, 26, 47, 51, 55, 58, 61  
 sf.colors, 22  
 sf.colors (plot), 21  
 sf\_extSoftVersion, 28  
 sfc, 28, 35, 47, 51, 55, 58  
 sgbp, 29  
 slice.sf (dplyr), 7  
 spread.sf (dplyr), 7  
 st, 29  
 st\_agr, 32  
 st\_agr<- (st\_agr), 32  
 st\_area (geos\_measures), 14  
 st\_as\_binary, 26, 28, 33, 38, 59  
 st\_as\_grob, 34  
 st\_as\_sf, 34, 60  
 st\_as\_sfc, 26, 27, 36, 40  
 st\_as\_text, 38  
 st\_bbox, 39  
 st\_bind\_cols (bind), 5  
 st\_boundary (geos\_unary), 17  
 st\_buffer (geos\_unary), 17  
 st\_cast, 5, 41, 44  
 st\_cast\_sfc\_default, 43  
 st\_centroid (geos\_unary), 17  
 st\_collection\_extract, 44  
 st\_combine, 8, 65  
 st\_combine (geos\_combine), 13  
 st\_contains, 56  
 st\_contains (geos\_binary\_pred), 11  
 st\_contains\_properly  
     (geos\_binary\_pred), 11  
 st\_convex\_hull (geos\_unary), 17  
 st\_coordinates, 45  
 st\_covered\_by, 56  
 st\_covered\_by (geos\_binary\_pred), 11  
 st\_covers, 56  
 st\_covers (geos\_binary\_pred), 11  
 st\_crosses, 56  
 st\_crosses (geos\_binary\_pred), 11  
 st\_crs, 46  
 st\_crs(), 37  
 st\_crs<- (st\_crs), 46  
 st\_difference, 14  
 st\_difference (geos\_binary\_ops), 10  
 st\_dimension, 15  
 st\_dimension (geos\_query), 16  
 st\_disjoint, 56  
 st\_disjoint (geos\_binary\_pred), 11  
 st\_distance (geos\_measures), 14  
 st\_drivers, 19, 20, 48, 69  
 st\_equals, 56  
 st\_equals (geos\_binary\_pred), 11  
 st\_equals\_exact, 56  
 st\_equals\_exact (geos\_binary\_pred), 11  
 st\_geohash, 49  
 st\_geometry, 50  
 st\_geometry<- (st\_geometry), 50  
 st\_geometry\_type, 51  
 st\_geometrycollection (st), 29  
 st\_graticule, 23, 52  
 st\_interpolate\_aw, 53

`st_intersection`, [14](#), [63](#)  
`st_intersection` (`geos_binary_ops`), [10](#)  
`st_intersects`, [33](#), [56](#)  
`st_intersects` (`geos_binary_pred`), [11](#)  
`st_is`, [54](#)  
`st_is_longlat`, [55](#)  
`st_is_simple` (`geos_query`), [16](#)  
`st_is_valid` (`geos_query`), [16](#)  
`st_is_within_distance`  
  (`geos_binary_pred`), [11](#)  
`st_jitter`, [55](#)  
`st_join`, [4](#), [56](#)  
`st_layers`, [57](#)  
`st_length` (`geos_measures`), [14](#)  
`st_line_merge` (`geos_unary`), [17](#)  
`st_line_sample`, [57](#)  
`st_linestring` (`st`), [29](#)  
`st_make_grid`, [58](#)  
`st_make_valid`, [59](#)  
`st_make_valid` (`valid`), [72](#)  
`st_multilinestring` (`st`), [29](#)  
`st_multipoint` (`st`), [29](#)  
`st_multipolygon` (`st`), [29](#)  
`st_node` (`geos_unary`), [17](#)  
`st_overlaps`, [56](#)  
`st_overlaps` (`geos_binary_pred`), [11](#)  
`st_point`, [35](#)  
`st_point` (`st`), [29](#)  
`st_point_on_surface` (`geos_unary`), [17](#)  
`st_polygon` (`st`), [29](#)  
`st_polygonize` (`geos_unary`), [17](#)  
`st_precision`, [59](#)  
`st_precision<-` (`st_precision`), [59](#)  
`st_proj_info` (`st_transform`), [65](#)  
`st_read`, [60](#)  
`st_read_db` (`st_read`), [60](#)  
`st_relate`, [12](#), [56](#), [62](#)  
`st_sample`, [63](#)  
`st_segmentize` (`geos_unary`), [17](#)  
`st_set_agr` (`st_agr`), [32](#)  
`st_set_crs` (`st_crs`), [46](#)  
`st_set_geometry` (`st_geometry`), [50](#)  
`st_set_precision` (`st_precision`), [59](#)  
`st_sf`, [6](#), [35](#)  
`st_sf` (`sf`), [26](#)  
`st_sfc` (`sfc`), [28](#)  
`st_simplify` (`geos_unary`), [17](#)  
`st_split`, [64](#)  
`st_sym_difference`, [14](#)  
`st_sym_difference` (`geos_binary_ops`), [10](#)  
`st_touches`, [56](#)  
`st_touches` (`geos_binary_pred`), [11](#)  
`st_transform`, [65](#)  
`st_triangulate` (`geos_unary`), [17](#)  
`st_union`, [4](#), [8](#), [11](#)  
`st_union` (`geos_combine`), [13](#)  
`st_viewport`, [67](#)  
`st_voronoi` (`geos_unary`), [17](#)  
`st_within`, [56](#)  
`st_within` (`geos_binary_pred`), [11](#)  
`st_wrap_dateline` (`st_transform`), [65](#)  
`st_write`, [59](#), [68](#)  
`st_write_db`, [59](#)  
`st_write_db` (`st_write`), [68](#)  
`st_zm`, [70](#)  
`summarise`, [42](#), [59](#)  
`summarise` (`dplyr`), [7](#)  
`summary.sfc`, [71](#)  
  
`t.sgbp` (`sgbp`), [29](#)  
`tibble`, [71](#)  
`transmute.sf` (`dplyr`), [7](#)  
`type_sum.sfc` (`tibble`), [71](#)  
  
`ungroup.sf` (`dplyr`), [7](#)  
`unit`, [34](#)  
`unite.sf` (`dplyr`), [7](#)  
  
`valid`, [72](#)  
`viewport`, [67](#)  
  
`write_sf`, [59](#)  
`write_sf` (`st_write`), [68](#)