Package ‘sfdep’

January 11, 2023

Title  Spatial Dependence for Simple Features
Version  0.2.3
Description  An interface to 'spdep' to integrate with 'sf' objects and the 'tidyverse'.
License  GPL-3
URL  https://sfdep.josiahparry.com,
    https://github.com/josiahparry/sfdep
Suggests  broom, dbscan, dplyr, ggplot2, knitr, magrittr, patchwork,
          purrr, pracma, rmarkdown, sfnetworks, stringr, testthat (>=
          3.0.0), tibble, tidyr, vctrs, yaml, zoo, Kendall, igraph,
          tidygraph
Config/testthat/edition  3
Encoding  UTF-8
RoxygenNote  7.2.3
Imports  sf, cli, spdep, stats, rlang
Depends  R (>= 3.5.0)
LazyData  true
VignetteBuilder  knitr
NeedsCompilation  no
Author  Josiah Parry [aut, cre] (<https://orcid.org/0000-0001-9910-865X>)
Maintainer  Josiah Parry <josiah.parry@gmail.com>
Repository  CRAN
Date/Publication  2023-01-11 06:30:02 UTC

R topics documented:

  active  ................................................................. 3
  as_sf  ............................................................... 4
  center_mean  ......................................................... 5
  complete_spacetime_cube  .......................................... 6
<table>
<thead>
<tr>
<th>R topics documented:</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond_permute_nb</td>
</tr>
<tr>
<td>critical_threshold</td>
</tr>
<tr>
<td>ellipse</td>
</tr>
<tr>
<td>emerging_hotspot_analysis</td>
</tr>
<tr>
<td>find_xj</td>
</tr>
<tr>
<td>global_c</td>
</tr>
<tr>
<td>global_colocation</td>
</tr>
<tr>
<td>global_c_perm</td>
</tr>
<tr>
<td>global_c_test</td>
</tr>
<tr>
<td>global_g_test</td>
</tr>
<tr>
<td>global_jc_perm</td>
</tr>
<tr>
<td>global_moran</td>
</tr>
<tr>
<td>global_moran_perm</td>
</tr>
<tr>
<td>global_moran_test</td>
</tr>
<tr>
<td>guerry</td>
</tr>
<tr>
<td>include_self</td>
</tr>
<tr>
<td>is_spacetime_cube</td>
</tr>
<tr>
<td>local_c</td>
</tr>
<tr>
<td>local_colocation</td>
</tr>
<tr>
<td>local_g</td>
</tr>
<tr>
<td>local_gstar</td>
</tr>
<tr>
<td>local_jc_bv</td>
</tr>
<tr>
<td>local_jc_uni</td>
</tr>
<tr>
<td>local_moran</td>
</tr>
<tr>
<td>local_moran_bv</td>
</tr>
<tr>
<td>losh</td>
</tr>
<tr>
<td>nb_match_test</td>
</tr>
<tr>
<td>nb_union</td>
</tr>
<tr>
<td>node_get_nbs</td>
</tr>
<tr>
<td>pairwise_colocation</td>
</tr>
<tr>
<td>pct_nonzero</td>
</tr>
<tr>
<td>recreate_listw</td>
</tr>
<tr>
<td>set_col</td>
</tr>
<tr>
<td>spacetime</td>
</tr>
<tr>
<td>spatial_gini</td>
</tr>
<tr>
<td>spt_update</td>
</tr>
<tr>
<td>std_dev_ellipse</td>
</tr>
<tr>
<td>std_distance</td>
</tr>
<tr>
<td>st_as_edges</td>
</tr>
<tr>
<td>st_as_graph</td>
</tr>
<tr>
<td>st_as_nodes</td>
</tr>
<tr>
<td>st_block_nb</td>
</tr>
<tr>
<td>st_cardinalties</td>
</tr>
<tr>
<td>st_complete_nb</td>
</tr>
<tr>
<td>st_contiguity</td>
</tr>
<tr>
<td>st_dist_band</td>
</tr>
<tr>
<td>st_inverse_distance</td>
</tr>
</tbody>
</table>
**active**

From a spacetime object, activate either the data or geometry contexts. The active object will then become available for manipulation.

**Usage**

```r
active(.data)
```

```r
activate(.data, what)
```

**Arguments**

- `.data` a spacetime object
- `.what` default NULL. Determines which context to activate. Valid argument values are "geometry" and "data". If left null, returns `.data`.

**Details**

A spacetime object contains both a data frame and an sf object. The data frame represents geographies over one or more time periods and the sf object contains the geographic information for those locations.

**Value**

For `activate()` an object of class spacetime with the specified context activated. `active()` returns a scalar character with the active context can be either "geometry" or "data".
Examples

```r
df_fp <- system.file("extdata", "bos-ecometric.csv", package = "sfdep")
geo_fp <- system.file("extdata", "bos-ecometric.geojson", package = "sfdep")

# read in data
df <- read.csv(
  df_fp, colClasses = c("character", "character", "integer", "double", "Date")
)
geo <- sf::st_read(geo_fp)

# Create spacetime object called 'bos'
bos <- spacetime(df, geo,
  .loc_col = ".region_id",
  .time_col = "time_period")

active(bos)
activate(bos, "geometry")
```

---

**as_sf**

*Cast between spacetime and sf classes*

---

**Description**

Cast between spacetime and sf classes

Convert sf object to spacetime

**Usage**

```r
as_sf(x, ...)
as_spacetime(x, .loc_col, .time_col, ...)
```

## S3 method for class 'sf'
as_spacetime(x, .loc_col, .time_col, ...)

**Arguments**

- `x` for `st_as_sf()` a spacetime object. For `as_spacetime()` an sf object.
- `...` arguments passed to merge.
- `.loc_col` the quoted name of the column containing unique location identifiers.
- `.time_col` the quoted name of the column containing time periods.

**Value**

For `as_spacetime()` returns a spacetime object. For `as_sf()`, an sf object.
center_mean

Examples

```r
df_fp <- system.file("extdata", "bos-econometric.csv", package = "sfdep")
geo_fp <- system.file("extdata", "bos-econometric.geojson", package = "sfdep")

# read in data
df <- read.csv(
  df_fp, colClasses = c("character", "character", "integer", "double", "Date")
)
geo <- sf::st_read(geo_fp)

# Create spacetime object called 'bos'
bos <- spacetime(df, geo,
  .loc_col = ".region_id",
  .time_col = "time_period")
as_sf(bos)
as_spacetime(as_sf(bos), ".region_id", "year")
```

center_mean  Calculate Center Mean Point

Description

Given an sfc object containing points, calculate a measure of central tendency.

Usage

```r
center_mean(geometry, weights = NULL)
center_median(geometry)
euclidean_median(geometry, tolerance = 1e-09)
```

Arguments

- `geometry`: an sfc object. If a polygon, uses `sf::st_point_on_surface()`.
- `weights`: an optional vector of weights to apply to the coordinates before calculation.
- `tolerance`: a tolerance level to terminate the process. This is passed to `pracma::geo_median()`.

Details

- `center_mean()` calculates the mean center of a point pattern
- `euclidean_median()` calculates the euclidean median center of a point pattern using the `pracma` package
- `center_median()` calculates the median center it is recommended to use the euclidean median over the this function.
Value

an sfc POINT object

See Also

Other point-pattern: `std_distance()`
Other point-pattern: `std_distance()`

Examples

```r
# Make a grid to sample from
grd <- sf::st_make_grid(n = c(1, 1), cellsize = c(100, 100), offset = c(0,0))

# sample 100 points
pnts <- sf::st_sample(grd, 100)

cm <- center_mean(pnts)
em <- euclidean_median(pnts)
cmed <- center_median(pnts)

plot(pnts)
plot(cm, col = "red", add = TRUE)
plot(em, col = "blue", add = TRUE)
plot(cmed, col = "green", add = TRUE)
```

---

**complete_spacetime_cube**

Convert spacetime object to spacetime cube

Description

Given a spacetime object, convert it to a spacetime cube. A spacetime cube ensures that there is a regular time-series for each geometry present.

Usage

```r
complete_spacetime_cube(x, ...)
```

Arguments

- `x` a spacetime object.
- `...` unused

Details

If observations are missing for a time period and location combination, columns will be populated with NAs.
See `is_spacetime_cube()` for more details on spacetime cubes.
cond_permute_nb

Value

A spacetime object that meets the criteria of spacetime cube.

Examples

df_fp <- system.file("extdata", "bos-ecometric.csv", package = "sfdep")
geo_fp <- system.file("extdata", "bos-ecometric.geojson", package = "sfdep")

# read in data
df <- read.csv(df_fp, colClasses = c("character", "character", "integer", "double", "Date"))
geo <- sf::st_read(geo_fp)

# Create spacetime object called `bos`
bos <- spacetime(df, geo,
    .loc_col = ".region_id",
    .time_col = "time_period")

# create a sample of data
set.seed(0)
sample_index <- sample(1:nrow(bos), nrow(bos) * 0.95)
incomplete_spt <- bos[sample_index,]

# check to see if is spacetime cube
is_spacetime_cube(incomplete_spt)

# complete it again
complete_spacetime_cube(incomplete_spt)

cond_permute_nb

Conditional permutation of neighbors

Description

Creates a conditional permutation of neighbors list holding i fixed and shuffling it’s neighbors.

Usage

cond_permute_nb(nb, seed = NULL)

Arguments

nb a neighbor list.

seed default null. A value to pass to set.seed() for reproducibility.

Value

A list of class nb where each element contains a random sample of neighbors excluding the observed region.
Examples

```r
nb <- st_contiguity(guerry)
nb[1:5]
# conditionally permute neighbors
perm_nb <- cond_permute_nb(nb)
perm_nb[1:5]
```

critical_threshold  

**Description**

Identifies the minimum distance in which each observation will have at least one neighbor.

**Usage**

```r
critical_threshold(geometry, k = 1)
```

**Arguments**

- `geometry`: an sf geometry column
- `k`: the minimum number of neighbors to check for

**Value**

a numeric scalar value.

**Examples**

```r
critical_threshold(sf::st_geometry(guerry))
```

ellipse  

**Description**

Generate an ellipse from center coordinates, major and minor axis radii, and angle rotation.

**Usage**

```r
ellipse(x = 0, y = 0, sx = 2, sy = 1, rotation = 0, n = 100)
```

```r
st_ellipse(geometry, sx, sy, rotation = 0, n = 100)
```
Arguments

- **x**: longitude of center point
- **y**: latitude of center point
- **sx**: radius of major axis
- **sy**: radius of minor axis
- **rotation**: the degree of rotation of the ellipse
- **n**: the number of coordinates to generate for the ellipse
- **geometry**: an sf ST_POINT geometry. Can be sfg, sfc, or sf object

Details

**ellipse()** returns a matrix of point locations defining the ellipse. **st_ellipse()** returns an sf object with LINE geography of the ellipse. Increasing **n** increases the number of points generated to define the ellipse shape.

**ellipse()** function is adapted from ggVennDiagram.

Value

- an sf object

Examples

```r
circle(n = 10)
circle(sf::st_point(c(0, 0)), sx = 10, sy = 10)
```

---

**emerging_hotspot_analysis**

*Emerging Hot Spot Analysis*

Description

Emerging Hot Spot Analysis identifies trends in spatial clustering over a period of time. Emerging hot spot analysis combines the Getis-Ord Gi* statistic with the Mann-Kendall trend test to determine if there is a temporal trend associated with local clustering of hot and cold spots.

Usage

```r
emerging_hotspot_analysis(
x,
.var,
k = 1,
include_gi = FALSE,
nb_col = NULL,
wt_col = NULL,
sim = 199,
```
threshold = 0.01,
...
)

Arguments

- **x**: a spacetime object and must be a spacetime cube see details for more.
- **.var**: a numeric vector in the spacetime cube with no missing values.
- **k**: default 1. The number of time lags to include in the neighborhood for calculating the local Gi*. See details for more.
- **include_gi**: default `FALSE`. If `TRUE`, includes the local Gi* calculations in the attribute `gi_star`.
- **nb_col**: Optional. Default `NULL`. The name of the column in the `geometry` context of `x` containing spatial neighbors. If `NULL`, Queen’s contiguity neighbors are identified.
- **wt_col**: Optional. Default `NULL`. The name of the column in the `geometry` context of `x` containing spatial weights. If `NULL`, row standardized weights are used.
- **nsim**: default 199. The number of simulations to run in calculating the simulated p-value for the local Gi*.
- **threshold**: default 0.01. The significance threshold to use.
- **...** unused.

Details

How Emerging Hot Spot Analysis Works:
Emerging Hot Spot Analysis is a somewhat simple process. It works by first calculating the Gi* statistic for each location in each time period (time-slice). Next, for each location across all time-periods, the Mann-Kendall trend test is done to identify any temporal trend in Gi* values over all time periods. Additionally, each location is classified into one of seventeen categories based on ESRI’s emerging hot spot classification criteria.

The Mann-Kendall trend test is done using `Kendall::MannKendall()`. Kendall is not installed with sfdep and should be installed prior to use.

Using your own neighbors and weights:
If you would like to use your own neighbors and weights, they must be created in the geometry context of a spacetime object. The arguments `nb_col` and `wt_col` must both be populated in order to use your own neighbor and weights definitions.

Time lagged neighbors:
In addition to identifying neighbors in space, emerging hotspot analysis also incorporates the same observations from k periods ago-called a time lag. If the time lag k is 1 and the unit of time is month, the neighbors for the calculation of Gi* would include the spatial neighbors’ values at time t and the same spatial neighbors’ values at time t-1. If k = 2, it would include t, t-1, and t-2.

Missing values:
Presently, there is no method of missing value handling. If there are missing values, the emerging hot spot analysis will fail. Be sure to fill or omit time-slices with missing values prior to using emerging hot spot analysis.
Value

Returns a data.frame.

See Also

How Emerging Hot Spot Analysis works, Emerging Hot Spot Analysis (Space Time Pattern Mining), and the video Spatial Data Mining II: A Deep Dive into Space-Time Analysis by ESRI.

Examples

df_fp <- system.file("extdata", "bos-ecometric.csv", package = "sfdep")
geo_fp <- system.file("extdata", "bos-ecometric.geojson", package = "sfdep")

# read in data
df <- read.csv(df_fp, colClasses = c("character", "character", "integer", "double", "Date"))
geo <- sf::st_read(geo_fp)

# Create spacetime object called `bos`
bos <- spacetime(df, geo,
   .loc_col = ".region_id",
   .time_col = "time_period")

# conduct EHSA
ehsa <- emerging_hotspot_analysis(
   x = bos,
   .var = "value",
   k = 1,
   nsim = 9
)
ehsa

---

**find_xj**

**Identify xj values**

Description

Find xj values given a numeric vector, x, and neighbors list, nb.

Usage

`find_xj(x, nb)`

Arguments

- **x**: a vector of any class
- **nb**: a nb object e.g. created by `st_contiguity()` or `st_knn()`
A list of length \( x \) where each element is a numeric vector with the same length as the corresponding element in \( \text{nb} \).

**Examples**

\[
\text{nb} \leftarrow \text{st_contiguity(} \text{sf::st.geometry(} \text{guerry)}\text{)}
\]

\[
\text{xj} \leftarrow \text{find_xj(} \text{guerry$crime_prop, nb)}
\]

\[
\text{xj}[1:3]
\]

---

**Description**

Compute Geary’s C

**Usage**

\[
\text{global_c(} x, \text{nb, wt, allow_zero = NULL)}
\]

**Arguments**

- **x**
  - A numeric vector.
- **nb**
  - A neighbor list object for example as created by \text{st_contiguity(}).
- **wt**
  - A weights list as created by \text{st_weights()}.
- **allow_zero**
  - If TRUE, assigns zero as lagged value to zone without neighbors.

**Value**

A list with two names elements \( C \) and \( K \) returning the value of Geary’s C and sample kurtosis respectively.

**See Also**

Other \text{global_c: global_c_perm()}, \text{global_c_test()}

**Examples**

\[
\text{nb} \leftarrow \text{guerry_nb$nb}
\]

\[
\text{wt} \leftarrow \text{guerry_nb$wt}
\]

\[
\text{x} \leftarrow \text{guerry_nb$crime_pers}
\]

\[
\text{global_c}(x, \text{nb, wt})
\]
Description

Calculate the Global Colocation Quotient (CLQ) for a categorical variable using simulation based significance testing.

Usage

```r
global_colocation(A, nb, nsim = 99)
```

Arguments

- `A`: a character or factor vector.
- `nb`: a neighbors list e.g. created by `st_knn()` or `st_contiguity()`
- `nsim`: default 99. An integer representing how many simulations to run for calculating the simulated p-values.

Details

**Definition:**
The CLQ is defined as $CLQ_{Global} = \frac{\sum_{A \in X} C_{A \rightarrow A}}{\sum_{A \in X} N_{A} \left( \frac{N_{A}}{N-1} \right)}$. The numerator identifies the observed proportion of same-category neighbors while the denominator contains the expected proportion of same-category neighbors under the assumption of no spatial association. Thus the CLQ is just a ratio of observed to expected.

**Inference:**
Inference is done using conditional permutation as suggested by Anselin 1995 where a number of replicates are created. The observed values are compared to the replicates and a the simulated p-value is the proportion of cases where the observed is more extreme as compared to replicate. The simulated p-value returns the lower p-value of either tail.

**Interpretation:**
Given that the CLQ is a ratio of the observed to expected, we interpret values larger than one to mean that there is more colocation than to be expected under the null hypothesis of no spatial association. When the value is smaller than 0, we interpret it to mean that there is less colocation than expected under the null.

**Value**

A list of two elements `CLQ` and `p_sim` containing the observed colocation quotient and the simulated p-value respectively.
References


Examples

```r
A <- guerry$main_city
nb <- st_contiguity(sf::st_geometry(guerry))
global_colocation(A, nb, 49)
```

---

**global_c_perm**

**Global C Permutation Test**

**Description**

Global C Permutation Test

**Usage**

```r
global_c_perm(
  x, nb, wt, nsim = 499,
  alternative = "greater",
  allow_zero = NULL,
  ...
)
```

**Arguments**

- `x` A numeric vector.
- `nb` a neighbor list object for example as created by `st_contiguity()`.
- `wt` a weights list as created by `st_weights()`.
- `nsim` number of simulations to run.
- `alternative` default "two.sided". Should be one of "greater", "less", or "two.sided" to specify the alternative hypothesis.
- `allow_zero` If TRUE, assigns zero as lagged value to zone without neighbors.
- `...` additional arguments passed to `spdep::geary.mc()`.

**Value**

an object of classes `htest` and `mc.sim`
**See Also**

Other global_c: `global_c_test()`, `global_c()`

**Examples**

```r
geo <- sf::st_geometry(guerry)
b <- st_contiguity(geo)
w <- st_weights(b)
x <- guerry$crime_pers
global_c_perm(x, b, w)
```

---

**Description**

Global C Test

**Usage**

```r
global_c_test(x, nb, wt, randomization = TRUE, allow_zero = NULL, ...)
```

**Arguments**

- `x`: A numeric vector.
- `nb`: a neighbor list object for example as created by `st_contiguity()`.
- `wt`: a weights list as created by `st_weights()`.
- `randomization`: default TRUE. Calculate variance based on randomization. If FALSE, under the assumption of normality.
- `allow_zero`: If TRUE, assigns zero as lagged value to zone without neighbors.
- `...`: additional arguments passed to `spdep::moran.mc()`

**Value**

an htest object

**See Also**

Other global_c: `global_c_perm()`, `global_c()`

**Examples**

```r
geo <- sf::st_geometry(guerry)
b <- st_contiguity(geo)
w <- st_weights(b)
x <- guerry$crime_pers
global_c_test(x, b, w)
```
global_g_test

Getis-Ord Global G

Description

Getis-Ord Global G

Usage

global_g_test(x, nb, wt, alternative = "greater", allow_zero = NULL, ...)

Arguments

x  A numeric vector.

nb  a neighbor list object for example as created by st_contiguity().

wt  a weights list as created by st_weights().

alternative  default "two.sided". Should be one of "greater", "less", or "two.sided" to specify the alternative hypothesis.

allow_zero  If TRUE, assigns zero as lagged value to zone without neighbors.

...  additional methods passed to spdep::globalG.test().

Value

an htest object

Examples

geo <- sf::st_geometry(guerry)
nb <- st_contiguity(geo)
wta <- st_weights(nb, style = "B")
x <- guerry$crime_pers
global_g_test(x, nb, wt)

---

global_jc_perm

Global Join Counts

Description

Calculate global join count measure for a categorical variable.
Usage

```r
global_jc_perm(fx, nb, wt, alternative = "greater", nsim = 499, allow_zero = FALSE, ...)
```

global_jc_test(fx, nb, wt, alternative = "greater", allow_zero = NULL, ...)

tally_jc(fx, nb, wt, allow_zero = TRUE, ...)

Arguments

- **fx**: a factor or character vector of the same length as nb.
- **nb**: a neighbor list object for example as created by `st_contiguity()`.
- **wt**: a weights list as created by `st_weights()`.
- **alternative**: default "two.sided". Should be one of "greater", "less", or "two.sided" to specify the alternative hypothesis.
- **nsim**: number of simulations to run.
- **allow_zero**: If TRUE, assigns zero as lagged value to zone without neighbors.
- **...**: additional arguments passed to methods

Details

- `global_jc_perm()` implements the monte-carlo based join count using `spdep::joincount.mc()`
- `global_jc_test()` implements the traditional BB join count statistic using `spdep::joincount.test()`
- `tally_jc()` calculated join counts for a variable `fx` and returns a data.frame using `spdep::joincount.multi()`

Value

An object of class `jclist` which is a list where each element is of class `htest` and `mc.sim`.

Examples

```r
geo <- sf::st_geometry(guerry)
b <- st_contiguity(geo)
w <- st_weights(nb, style = "B")
f <- guerry$region
global_jc_perm(f, nb, w)

global_jc_test(f, nb, w)

tally_jc(f, nb, w)
```
**global_moran**  
*Calculate Global Moran’s I*

**Description**  
Calculate Global Moran’s I

**Usage**  
`global_moran(x, nb, wt, na_ok = FALSE, ...)`

**Arguments**  
- `x`: A numeric vector.
- `nb`: a neighbor list object for example as created by `st_contiguity()`.
- `wt`: a weights list as created by `st_weights()`.
- `na_ok`: default FALSE. If FALSE presence or NA or Inf results in an error.
- `...`: additional arguments passed to `spdep::moran()`.

**Value**  
an `htest` object

**See Also**  
Other `global_moran`: `global_moran_bv()`, `global_moran_perm()`, `global_moran_test()`, `local_moran_bv()`

**Examples**
```r
nb <- guerry_nb$nb
wt <- guerry_nb$wt
x <- guerry_nb$crime_pers
moran <- global_moran(x, nb, wt)
```

---

**global_moran_bv**  
*Compute the Global Bivariate Moran’s I*

**Description**  
Given two continuous numeric variables, calculate the bivariate Moran’s I. See details for more.

**Usage**  
`global_moran_bv(x, y, nb, wt, nsim = 99, scale = TRUE)`
Arguments

- **x**: A numeric vector of same length as \( y \).
- **y**: A numeric vector of same length as \( x \).
- **nb**: A neighbor list object for example as created by `st_contiguity()`.
- **wt**: A weights list as created by `st_weights()`.
- **nsim**: The number of simulations to run.
- **scale**: Default `TRUE`.

Details

The Global Bivariate Moran is defined as

\[
I_B = \frac{\Sigma_i (\Sigma_j w_{ij} y_j x_i)}{\Sigma_i x_i^2}
\]

It is important to note that this is a measure of autocorrelation of \( X \) with the spatial lag of \( Y \). As such, the resultant measure may overestimate the amount of spatial autocorrelation which may be a product of the inherent correlation of \( X \) and \( Y \).

Value

An object of class `boot`.

References

Global Spatial Autocorrelation (2): Bivariate, Differential and EB Rate Moran Scatter Plot, Luc Anselin

See Also

Other `global_moran`: `global_moran_perm()`, `global_moran_test()`, `global_moran()`, `local_moran_bv()`

Examples

```r
x <- guerry_nb$crime_pers
y <- guerry_nb$wealth
nb <- guerry_nb$nb
wt <- guerry_nb$wt
global_moran_bv(x, y, nb, wt)
```
Global Moran Permutation Test

Usage

`global_moran_perm(x, nb, wt, alternative = "two.sided", nsim = 499, ...)`

Arguments

- `x`: A numeric vector.
- `nb`: A neighbor list object for example as created by `st_contiguity()`.
- `wt`: A weights list as created by `st_weights()`.
- `alternative`: Default "two.sided". Should be one of "greater", "less", or "two.sided" to specify the alternative hypothesis.
- `nsim`: Number of simulations to run.
- `...`: Additional arguments passed to `spdep::moran.mc()`.

Value

An object of classes `htest`, and `mc.sim`.

See Also

Other `global_moran`: `global_moran_bv()`, `global_moran_test()`, `global_moran()`, `local_moran_bv()`.

Examples

```r
nb <- guerry_nb$nb
wt <- guerry_nb$wt
x <- guerry_nb$crime_pers
moran <- global_moran_perm(x, nb, wt)
broom::tidy(moran)
```
global_moran_test

Global Moran Test

Description

Global Moran Test

Usage

global_moran_test(
  x, nb, wt,
  alternative = "greater",
  randomization = TRUE,
  ...
)

Arguments

x A numeric vector.

nb a neighbor list object for example as created by st_contiguity().

wt a weights list as created by st_weights().

alternative default "two.sided". Should be one of "greater", "less", or "two.sided" to specify the alternative hypothesis.

randomization default TRUE. Calculate variance based on randomization. If FALSE, under the assumption of normality.

... additional arguments passed to spdep::moran.mc()

Value

an object of class htest

See Also

Other global_moran: global_moran_bv(), global_moran_perm(), global_moran(), local_moran_bv()

Examples

nb <- guerry_nb$nb
wt <- guerry_nb$wt
x <- guerry_nb$crime_pers
global_moran_test(x, nb, wt)
Description

This dataset has been widely used to demonstrate geospatial methods and techniques. As such it is useful for inclusion to this R package for the purposes of example. The dataset in this package is modified from Guerry by Michael Friendly.

Usage

```r
guerry

guerry_nb
```

Format

An object of class `sf` (inherits from `tbl_df`, `tbl`, `data.frame`) with 85 rows and 27 columns.

- `guerry` an sf object with 85 observations and 27 variables. `guerry_nb` has 2 additional variables created by `sfdep`.

Details

`guerry` and `guerry_nb` objects are sf class objects. These are polygons of the boundaries of France (excluding Corsica) as they were in 1830.

Source

`Guerry::gfrance85`

Description

Includes observed region in list of own neighbors. For some neighbor lists, it is important to include the ith observation (or self) in the neighbors list, particularly for kernel weights.

Usage

```r
include_self(nb)

remove_self(nb)
```
is_spacetime_cube

Arguments

  nb        an object of class nb e.g. made by st_contiguity()

Value

An object of class nb.

Examples

  nb <- st_contiguity(guerry)
  self_included <- include_self(nb)
  self_included
  remove_self(self_included)

is_spacetime_cube  Test if a spacetime object is a spacetime cube

Description

Given an object with class spacetime, determine if it is a spacetime cube. If the time-series is is irregular a warning is emitted (see validate_spacetime() for more on the restrictions on the time column.

Usage

  is_spacetime_cube(x, ...)

Arguments

  x        a spacetime object
  ...      unused

Details

A spacetime object is a spacetime cube when it contains a regular time-series representation of each geometry. That is, only one observation for at each time period per geography is present.

The number of rows in a spacetime cube is the number of geographies multiplied by the number of time periods. For example if there are 10 locations and 20 time periods, the number of rows must be 200.

Value

A logical scalar.
Validation

`is_spacetime_cube()` runs a number of checks that to ensure that the provided object is in fact a spacetime cube. It checks that:

- the number of rows is equal to the number of locations multiplied by the number of time periods
- each time period has an equal number of observations
- each location has an equal number of observations
- each combination of time period and location has only one observation
- that the time-series is regular

Examples

```r
df_fp <- system.file("extdata", "bos-ecometric.csv", package = "sfdep")
geo_fp <- system.file("extdata", "bos-ecometric.geojson", package = "sfdep")

# read in data
df <- read.csv(df_fp, colClasses = c("character", "character", "integer", "double", "Date"))
geo <- sf::st_read(geo_fp)

# Create spacetime object called 'bos'
bos <- spacetime(df, geo,
    .loc_col = ".region_id",
    .time_col = "time_period")

is_spacetime_cube(bos)
```

local_c

**Compute Local Geary statistic**

**Description**

The Local Geary is a local adaptation of Geary’s C statistic of spatial autocorrelation. The Local Geary uses squared differences to measure dissimilarity unlike the Local Moran. Low values of the Local Geary indicate positive spatial autocorrelation and large refers to negative spatial autocorrelation. Inference for the Local Geary is based on a permutation approach which compares the observed value to the reference distribution under spatial randomness. The Local Geary creates a pseudo p-value. This is not an analytical p-value and is based on the number of permutations and as such should be used with care.

**Usage**

`local_c(x, nb, wt, ...)`

`local_c_perm(x, nb, wt, nsim = 499, alternative = "two.sided", ...)`
Arguments

- **x**: a numeric vector, or list of numeric vectors of equal length.
- **nb**: a neighbor list
- **wt**: a weights list
- **...**: other arguments passed to `spdep::localC_perm()`, e.g. `zero.policy = TRUE` to allow for zones without neighbors.
- **nsim**: the number of simulations used to generate reference distribution.
- **alternative**: a character defining the alternative hypothesis. Must be one of "two.sided", "less" or "greater".

Details

Overview:
The Local Geary can be extended to a multivariate context. When `x` is a numeric vector, the univariate Local Geary will be calculated. To calculate the multivariate Local Moran provide either a list or a matrix. When `x` is a list, each element must be a numeric vector of the same length and of the same length as the neighbours in `listw`. In the case that `x` is a matrix the number of rows must be the same as the length of the neighbours in `listw`.

While not required in the univariate context, the standardized Local Geary is calculated. The univariate Local Geary is always standardized.

The univariate Local Geary is calculated as $c_i = \sum_j w_{ij} (x_i - x_j)^2$ and the multivariate Local Geary is calculated as $c_{k,i} = \sum_{v=1}^{k} c_{v,i}$ as described in Anselin (2019).

Implementation:
These functions are based on the implementations of the local Geary statistic in the development version of spdep. They are based on `spdep::localC` and `spdep::localC_perm`. `spdep::localC_perm` and thus `local_c_perm` utilize a conditional permutation approach to approximate a reference distribution where each observation $i$ is held fixed, randomly samples neighbors, and calculated the local C statistic for that tuple ($c_i$). This is repeated $nsim$ times. From the simulations 3 different types of p-values are calculated—all of which have their potential flaws. So be extra judicious with using p-values to make conclusions.

- **p_ci**: utilizes the sample mean and standard deviation. The p-value is then calculated using `pnorm()`—assuming a normal distribution which isn’t always true.
- **p_ci_sim**: uses the rank of the observed statistic.
- **p_folded_sim**: follows the pysal implementation where p-values are in the range of [0, 0.5]. This excludes 1/2 of all p-values and should be used with caution.

Value

A data frame with columns

- **ci**: Local Geary statistic
- **e_ci**: expected value of the Local Geary based on permutations
- **z_ci**: standard deviation based on permutations
- **var_ci**: variance based on permutations
local_colocation

Description

The local indicator of the colocation quotient (LCLQ) is a Local Indicator of Spatial Association (LISA) that evaluates if a given observation’s subcategory in A is colocated with subcategories in B. Like the CLQ, the LCLQ provides insight into the asymmetric relationships between subcategories of A and B (where B can also equal A) but at the local level.

The LCLQ is defined using Gaussian kernel weights and an adaptive bandwidth (see `st_kernel_weights()`). However, any type of weights list can be used. Kernel weights are used to introduce a decay into the calculation of the CLQ. This ensures that points nearer to the focal point have more influence than those that are more distant.

Usage

```
local_colocation(A, B, nb, wt, nsim)
```
Arguments

A a character or factor vector.
B a character or factor vector.
nb a neighbors list e.g. created by \texttt{st_knn()} or \texttt{st_contiguity()}
wt a weights list. Recommended that it is a Gaussian kernel weights list using an adaptive bandwidth e.g. created by \texttt{st_kernel_weights(nb, geometry, "gaussian", adaptive = TRUE)} that does not include the self.
nsim default 99. An integer representing how many simulations to run for calculating the simulated p-values.

Details

The LCLQ is defined as $LCLQ_{A_i \rightarrow B} = \frac{N_{A_i \rightarrow B}}{N/(N-1)}$ where $N_{A_i \rightarrow B} = \sum_{j=1}^{N} w_{ij} f_{ij} \left( \frac{\sum_{j=1}^{N} w_{ij}}{\sum_{j=1}^{N} w_{ij}} \right)$.
And the weights matrix, $w_{ij}$, uses adaptive bandwidth Gaussian kernel weights.

LCLQ is only calculated for those subcategories which are present in the neighbor list. If a subcategory is not present, then the resultant LCLQ and simulated p-value will be NA.

Value

d a data frame with as many rows as observations in A and two times as many columns as unique values in B. Columns contain each unique value of B as well as the simulated p-value for each value of B.

References


Examples

A <- guerry$main_city
B <- guerry$region
geo <- sf::st_centroid(sf::st_geometry(guerry))
nb <- include_self(st_knn(geo, 5))
wt <- st_kernel_weights(nb, geo, "gaussian", adaptive = TRUE)
res <- local_colocation(A, B, nb, wt, 9)
tail(res)
local_g

Local G

Description

Calculate the local Geary statistic for a given variable.

Usage

local_g(x, nb, wt, alternative = "two.sided", ...)
local_g_perm(x, nb, wt, nsim = 499, alternative = "two.sided", ...)

Arguments

- **x**: A numeric vector.
- **nb**: a neighbor list object for example as created by `st_contiguity()`.
- **wt**: a weights list as created by `st_weights()`.
- **alternative**: default "two.sided". Should be one of "greater", "less", or "two.sided" to specify the alternative hypothesis.
- **...**: methods passed to `spdep::localG()` or `spdep::localG_perm()`
- **nsim**: The number of simulations to run.

Value

a data.frame with columns:

- **gi**: the observed statistic
- **e_gi**: the permutation sample mean
- **var_gi**: the permutation sample variance
- **p_value**: the p-value using sample mean and standard deviation
- **p_folded_sim**: p-value based on the implementation of Pysal which always assumes a two-sided test taking the minimum possible p-value
- **skewness**: sample skewness
- **kurtosis**: sample kurtosis

Examples

```r
x <- guerry$crime_pers
nb <- st_contiguity(guerry)
wt <- st_weights(nb)
res <- local_g_perm(x, nb, wt)
head(res)
```
local_gstar Local G*

Description
Calculate the local Gi* statistic.

Usage
local_gstar(x, nb, wt, alternative = "two.sided", ...)

local_gstar_perm(x, nb, wt, nsim = 499, alternative = "two.sided", ...)

Arguments
x A numeric vector.

nb a neighbor list object for example as created by st_contiguity().

wt a weights list as created by st_weights().

alternative default "two.sided". Should be one of "greater", "less", or "two.sided" to specify the alternative hypothesis.

... methods passed to spdep::localG() or spdep::localG_perm()

nsim The number of simulations to run.

Value
a data.frame with columns:
• gi: the observed statistic
• e_gi: the permutation sample mean
• var_gi: the permutation sample variance
• p_value: the p-value using sample mean and standard deviation
• p_folded_sim: p-value based on the implementation of Pysal which always assumes a two-sided test taking the minimum possible p-value
• skewness: sample skewness
• kurtosis: sample kurtosis

Examples
nb <- st_contiguity(guerry)
wt <- st_weights(nb)
x <- guerry$crime_pers

res <- local_gstar_perm(x, nb, wt)
head(res)

res <- local_gstar(x, nb, wt)
head(res)
local_jc_bv  \hspace{1cm} \textit{Bivariate local join count}

\textbf{Description}

Bivariate local join count

\textbf{Usage}

\texttt{local_jc_bv(x, z, nb, wt, nsim = 499)}

\textbf{Arguments}

- \texttt{x} \hspace{1cm} a binary variable either numeric or logical
- \texttt{z} \hspace{1cm} a binary variable either numeric or logical
- \texttt{nb} \hspace{1cm} a neighbors list object.
- \texttt{wt} \hspace{1cm} default \texttt{st_weights(nb, style = \textquoteleft B\textquoteright)}. A binary weights list as created by \texttt{st_weights(nb, style = \textquoteleft B\textquoteright)}.
- \texttt{nsim} \hspace{1cm} the number of conditional permutation simulations

\textbf{Value}

\texttt{a data.frame} with two columns \texttt{join_count} and \texttt{p_sim} and number of rows equal to the length of arguments \texttt{x}, \texttt{z}, \texttt{nb}, and \texttt{wt}.

\textbf{Examples}

\begin{verbatim}
x <- as.integer(guerry$infants > 23574)
z <- as.integer(guerry$donations > 10973)
nb <- st_contiguity(guerry)
wt <- st_weights(nb, style = \textquoteleft B\textquoteright)
local_jc_bv(x, z, nb, wt)
\end{verbatim}

local_jc_uni  \hspace{1cm} \textit{Compute local univariate join count}

\textbf{Description}

The univariate local join count statistic is used to identify clusters of rarely occurring binary variables. The binary variable of interest should occur less than half of the time.
local_jc_uni

Usage

local_jc_uni(
    fx,
    chosen,
    nb,
    wt = st_weights(nb, style = "B"),
    nsim = 499,
    alternative = "two.sided",
    iseed = NULL
)

Arguments

- **fx**: a binary variable either numeric or logical
- **chosen**: a scalar character containing the level of `fx` that should be considered the observed value (1).
- **nb**: a neighbors list object.
- **wt**: default `st_weights(nb, style = "B")`. A binary weights list as created by `st_weights(nb, style = "B")`.
- **nsim**: the number of conditional permutation simulations
- **alternative**: default "greater". One of "less" or "greater".
- **iseed**: default NULL, used to set the seed for possible parallel RNGs

Details

The local join count statistic requires a binary weights list which can be generated with `st_weights(nb, style = "B")`. Additionally, ensure that the binary variable of interest is rarely occurring in no more than half of observations.

P-values are estimated using a conditional permutation approach. This creates a reference distribution from which the observed statistic is compared. For more see Geoda Glossary. Calls `spdep::local_joincount_uni()`.

Value

-a data.frame with two columns `join_count` and `p_sim` and number of rows equal to the length of arguments `x`, `nb`, and `wt`.

Examples

```r
res <- dplyr::transmute(
    guerry,
    top_crime = as.factor(crime_prop > 9000),
    nb = st_contiguity(geometry),
    wt = st_weights(nb, style = "B"),
    jc = local_jc_uni(top_crime, "TRUE", nb, wt))
tidyrr::unnest(res, jc)
```
local_moran

Calculate the Local Moran's I Statistic

Description
Moran's I is calculated for each polygon based on the neighbor and weight lists.

Usage
local_moran(x, nb, wt, alternative = "two.sided", nsim = 499, ...)

Arguments
- **x**: A numeric vector.
- **nb**: a neighbor list object for example as created by st_contiguity().
- **wt**: a weights list as created by st_weights().
- **alternative**: default "two.sided". Should be one of "greater", "less", or "two.sided" to specify the alternative hypothesis.
- **nsim**: The number of simulations to run.
- **...**: See ?spdep::localmoran_perm() for more options.

Details
local_moran() calls spdep::localmoran_perm() and calculates the Moran I for each polygon. As well as provide simulated p-values.

Value
a data.frame containing the columns ii, eii, var_ii, z_ii, p_ii, p_ii_sim, and p_folded_sim. For more details please see spdep::localmoran_perm().

See Also
Other stats: st_lag()

Examples
library(magrittr)
lisa <- guerry %>%
dplyr::mutate(nb = st_contiguity(geometry),
              wt = st_weights(nb),
              moran = local_moran(crime_pers, nb, wt))

# unnest the dataframe column
tidyrr::unnest(lisa, moran)
local_moran_bv

Compute the Local Bivariate Moran’s I Statistic

Description

Given two continuous numeric variables, calculate the bivariate Local Moran’s I.

Usage

local_moran_bv(x, y, nb, wt, nsim = 499)

Arguments

x a numeric vector of same length as y.
y a numeric vector of same length as x.
nb a neighbor list object for example as created by st_contiguity().
wt a weights list as created by st_weights().
nsim the number of simulations to run.

Details

The Bivariate Local Moran, like its global counterpart, evaluates the value of x at observation i with its spatial neighbors’ value of y. The value of

\[ I_i^B \]

is just \( x_i \times Wy_i \). Or, in simpler words the local bivariate Moran is the result of multiplying x by the spatial lag of y. Formally it is defined as

\[ I_i^B = cx_i \sum w_{ij} y_j \]

Value

a data.frame containing two columns Ib and p_sim containing the local bivariate Moran’s I and simulated p-values respectively.

References

Local Spatial Autocorrelation (3): Multivariate Local Spatial Autocorrelation, Luc Anselin

See Also

Other global_moran: global_moran_bv(), global_moran_perm(), global_moran_test(), global_moran()
Examples

```r
x <- guerry_nb$crime_pers
y <- guerry_nb$wealth
nb <- guerry_nb$nb
wt <- guerry_nb$wt
local_moran_bv(x, y, nb, wt)
```

**losh**

*Local spatial heteroscedacity*

**Description**

Local spatial heteroscedacity

**Usage**

```r
losh(x, nb, wt, a = 2, ...)
losh_perm(x, nb, wt, a = 2, nsim = 499, ...)
```

**Arguments**

- **x**: a numeric vector.
- **nb**: a neighbor list for example created by `st_contiguity()`
- **wt**: a weights list for example created by `st_weights()`
- **a**: the exponent applied to the local residuals
- **...**: methods passed to `spdep::LOSH`
- **nsim**: number of simulations to run

**Value**

a data.frame with columns

- hi: the observed statistic
- e_hi: the sample average
- var_hi: the sample variance
- z_hi: the approximately Chi-square distributed test statistic
- x_bar_i: the local spatially weight mean for observation i
- ei: residuals

**Examples**

```r
nb <- st_contiguity(guerry)
wt <- st_weights(nb)
x <- guerry$crime_pers
losh(x, nb, wt)
losh(x, nb, wt)
losh_perm(x, nb, wt, var_hi = FALSE)
losh_perm(x, nb, wt, nsim = 49)
```
Local Neighbor Match Test

**Description**

Implements the Local Neighbor Match Test as described in *Tobler’s Law in a Multivariate World* (Anselin and Li, 2020).

**Usage**

```r
nb_match_test(
  x,
  nb,
  wt = st_weights(nb),
  k = 10,
  nsim = 499,
  scale = TRUE,
  .method = "euclidian",
  .p = 2
)
```

**Arguments**

- `x` a numeric vector or a list of numeric vectors of equal length.
- `nb` a neighbor list object for example as created by `st_contiguity()`.
- `wt` a weights list as created by `st_weights()`.
- `k` the number of neighbors to identify in attribute space. Should be the same as number of neighbors provided in `st_knn`.
- `nsim` the number of simulations to run for calculating the simulated p-value.
- `scale` default `TRUE`. Whether `x` should be scaled or not. Note that measures should be standardized.
- `method` default "euclidian". The distance measure passed to `stats::dist()`.
- `p` default 2. The power of Minkowski distance passed to the `p` argument in `stats::dist()`.

**Value**

a `data.frame` with columns

- `n_shared` (integer): the number of shared neighbors between geographic and attribute space
- `nb_matches` (list): matched neighbor indexes. Each element is an integer vector of same length as the ith observation of `n_shared`
- `knn_nb` (list): the neighbors in attribute space
- `probability` (numeric): the geometric probability of observing the number of matches
- `p_sim` (numeric): a folded simulated p-value
Examples

```r
library(magrittr)
guerry %>%
dplyr::transmute(nb = st_knn(geometry, k = 10),
                nmt = nb_match_test(list(crime_pers, literacy, suicides),
                                    nb, nsim = 999)) %>%
tidyr::unnest(nmt)
```

---

Set Operations

Description

Perform set operations element-wise on two lists of equal length.

Usage

```r
nb_union(x, y)
nb_intersect(x, y)
nb_setdiff(x, y)
```

Arguments

- `x`: list of class `nb`
- `y`: list of class `nb`

Details

- `nb_union()` returns the union of elements in each element of `x` and `y`
- `nb_intersect()` returns the intersection of elements in each element of `x` and `y`
- `nb_setdiff()` returns the intersection of elements in each element of `x` and `y`

Value

A list of class `nb`

Examples

```r
nb <- st_contiguity(guerry$geometry)
nb_knn <- st_knn(guerry$geometry, k = 3)
nb_setdiff(nb, nb_knn)
nb_union(nb, nb_knn)
nb_intersect(nb, nb_knn)
```
node_get_nbs

Create node features from edges

Description

Given a tidygraph object, create a list column of edge data for each node in the node context.

Usage

node_get_nbs()
node_get_edge_list()
node_get_edge_col(edges, .var)

Arguments

edges an edge list as created by node_get_edge_list()
.var the quoted name of a column in the edge context.

Details

- node_get_nbs(): creates a neighbor list in the nodes context based on the adjacency list. This returns a nb class object with the neighboring nodes.
  - Uses igraph::get.adjlist()
- node_get_edge_list(): creates an edge list. The edge list contains the row index of the edge relationships in the edge context for each node.
  - Uses igraph::get.adjedgelist().
- node_get_edge_col(): creates a list column containing edge attributes as a list column in the node context (much like find_xj()).
  - Uses igraph::get.edge.attribute()

Value

A list column

Examples

if (interactive()) {
  net <- sfnetworks::as_sfnetwork(
    sfnetworks::roxel
  )

dplyr::mutate(
  net,
```r
nb = node_get_nbs(),
edges = node_get_edge_list(),
types = node_get_edge_col(edges, "type")
}
}

pairwise_colocation  \textit{Pairwise Colocation Quotient}

**Description**

Calculate the pairwise colocation quotient (CLQ) for two categorical variables using conditional permutation.

**Usage**

```r
pairwise_colocation(A, B, nb, nsim = 99)
```

**Arguments**

- `A`: a character or factor vector.
- `B`: a character or factor vector.
- `nb`: a neighbors list e.g. created by `st_knn()` or `st_contiguity()`
- `nsim`: default 99. An integer representing how many simulations to run for calculating the simulated p-values.

**Details**

**Intuition:**

The pairwise CLQ is used to test if there is a spatial directional association between subcategories of two vectors `A` and `B`. Compared to the cross-K metric and the join count statistic, the pairwise CLQ can elucidate the presence of an asymmetric relationship between subcategories of `A` and `B`. `A` and `B` can either be separate categorical vectors or the same categorical vector.

"The null hypothesis for a CLQ-based analysis is 'given the clustering of the joint population, there is no spatial association between pairs of categorical subsets.'"

**Definition:**

The pairwise colocation quotient is defined as "the ratio of observed to expected proportions of `B` among `A`'s nearest neighbors. Formally this is given by \( CLQ_{A \rightarrow B} = \frac{C_{A \rightarrow B}}{N_{B}/(N-1)} \) where \( C_{A \rightarrow B} = \sum_{i=1}^{N_A} \sum_{j=1}^{v} \frac{B_{ij}(1.0)}{v} \)."

**Inference:**

Inference is done using conditional permutation as suggested by Anselin 1995 where a number of replicates are created. The observed values are compared to the replicates and a the simulated p-value is the proportion of cases where the observed is more extreme as compared to replicate. The simulated p-value returns the lower p-value of either tail.
**Interpretation:**
Given that the CLQ is a ratio of the observed to expected, we interpret values larger than one to mean that there is more colocation than to be expected under the null hypothesis of no spatial association. When the value is smaller than 0, we interpret it to mean that there is less colocation than expected under the null.

**Value**
A matrix where the rownames are the unique values of A and the column names are the unique values of B and their simulated p-values in the form of \( p_{\text{sim}}(B) \).

**Examples**
```r
A <- guerry$main_city
B <- guerry$region
nb <- st_knn(sf::st_geometry(guerry), 5)
pairwise_colocation(B, A, nb)
pairwise_colocation(B, B, nb, 199)
```

---

## pct_nonzero

<table>
<thead>
<tr>
<th>pct_nonzero</th>
<th>Percent Non-zero Neighbors</th>
</tr>
</thead>
</table>

**Description**
Calculate the percentage of non-zero neighbors in a neighbor list.

**Usage**
```r
pct_nonzero(nb)
```

**Arguments**
- `nb`: a neighbor list object

**Value**
a scalar double

**Examples**
```r
nb <- st_contiguity(guerry)
pct_nonzero(nb)
```
recreate_listw

Create a listw object from a neighbors and weight list

Description

Given a neighbor and weight list, create a listw object.

Usage

recreate_listw(nb, wt)

Arguments

- **nb**: a neighbor list object for example as created by `st_contiguity()`.
- **wt**: a weights list as created by `st_weights()`.

Value

a listw object

Examples

recreate_listw(guerry_nb$nb, guerry_nb$wt)

---

set_col

Set columns from geometry to data

Description

Set a column from the geometry context of a spacetime object to the data context.

Usage

set_col(x, .from_geo, .to_data = .from_geo)

set_wts(x, .wt_col = "wt")

set_nbs(x, .nb_col = "nb")

Arguments

- **x**: a spacetime object
- **.from_geo**: the name of the column in the geometry context
- **.to_data**: the name of the new variable to create in the data context
- **.wt_col**: the name of the weights column in the geometry context
- **.nb_col**: the name of neighbor column in the geometry context
Details

These functions will reorder the spacetime object to ensure that it is ordered correctly based on the location time columns in the geometry context defined by the \texttt{loc\_col} and \texttt{time\_col} attributes respectively.

\texttt{set\_wts()} and \texttt{set\_nbs()} create a new column in the data context with the same name as the column in the geometry context. If a different name is desired use \texttt{set\_col()}

Value

A spacetime object with an active data context and a new column from the geometry context.

Examples

\begin{verbatim}
df_fp <- system.file("extdata", "bos-ecometric.csv", package = "sfdep")
geo_fp <- system.file("extdata", "bos-ecometric.geojson", package = "sfdep")

# read in data
df <- read.csv(
  df_fp, colClasses = c("character", "character", "integer", "double", "Date")
)
geo <- sf::st_read(geo_fp)

# Create spacetime object called `bos`
bos <- spacetime(df, geo,
  .loc\_col = ".region\_id",
  .time\_col = "time\_period")
bos <- activate(bos, "geometry")
bos$nb <- st\_contiguity(bos)
bos$wt <- st\_weights(bos$nb)
bos$card <- st\_cardinalties(bos$nb)

set\_nbs(bos)
set\_wts(bos)
set\_col(bos, "card")
set\_col(bos, "card", "cardinalities")
\end{verbatim}

\textit{spacetime} \hspace{4cm} \textit{Construct a spacetime object}

Description

A spacetime object is a collection of a linked data frame and an sf objects. It can be thought of as geography linked to a table that represents those geographies over one or more time periods.
Usage

spacetime(.data, .geometry, .loc_col, .time_col, active = "data")

new_spacetime(.data, .geometry, .loc_col, .time_col, active = "data")

validate_spacetime(.data, .geometry, .loc_col, .time_col)

is_spacetime(x, ...)

is.spacetime(x, ...)

Arguments

.data an object with base class data.frame containing location and time identifiers .loc_col and .time_col respectively.

.geometry an sf object with columns .loc_col and .time_col

.loc_col the quoted name of the column containing unique location identifiers. Must be present in both .data and .geometry.

.time_col the quoted name of the column containing time periods must be present .data. See details for more

.active default "data". The object to make active. See activate() for more.

.x an object to test

... unused

Details

Create a spacetime representation of vector data from a data.frame and an sf object with spacetime(). .time_col must be able to be sorted. As such, .time_col cannot be a character vector. It must have a base type of (typeof()) either double or integer—the case in dates or factors respectively. An edge case exists with POSIXlt class objects as these can be sorted appropriately but have a base type of list.

spacetime() is a wrapper around new_spacetime(). Spacetimes are validated before creation with validate_spacetime().

Check if an object is a spacetime object with is_spacetime() or is.spacetime().

Value

- spacetime() and new_spacetime() construct spacetime class objects
- validate_spacetime() returns nothing but will elicit a warning or error if the spacetime object is not validly constructed
- is_spacetime() and is.spacetime() return a logical scalar indicating if an object inherits the spacetime class
Validation

validate_spacetime() checks both .data and .geometry to ensure that the constructed space-time object meets minimum requirements:

- .data inherits the data.frame class
- .geometry is an sf object
- ensures that .time_col is of the proper class
- ensures there are no missing geometries in .geometry
- checks for duplicate geometries
- ensures .loc_col are the same type in .data and .geometry
- lastly informs of missing values in additional columns in .data

Examples

df_fp <- system.file("extdata", "bos-ecometric.csv", package = "sfdep")
geo_fp <- system.file("extdata", "bos-ecometric.geojson", package = "sfdep")

# read in data
df <- read.csv(
  df_fp, colClasses = c("character", "character", "integer", "double", "Date")
)

go <- sf::st_read(geo_fp)

bos <- spacetime(df, geo, ".region_id", "year")
is_spacetime(bos)
bos

---

**spatial_gini**  

**Spatial Gini Index**

Description

Calculates the spatial Gini index for a given numeric vector and neighbor list. Based on the formula provided Rey and Smith (2013).

Usage

spatial_gini(x, nb)

Arguments

- `x` a numeric vector without missing values
- `nb` a neighbor list, for example created with `st_contiguity()`
Details

The Gini index is a global measure of inequality based on the Lorenz curve. It ranges between 0 and 1 where 0 is perfect equality and 1 is perfect inequality.

The spatial Gini index decomposes the Gini coefficient based on spatial neighbors.

Value

A data frame with columns:

- G: the Gini index
- NBG: the neighbor composition of the Gini coefficient
- NG: the non-neighbor composition of the Gini coefficient
- SG: the Spatial Gini which is equal to \( NG \times \frac{1}{G} \)

References

doi: 10.1007/s120760120086z

Examples

```r
nb <- st_contiguity(guerry)
x <- guerry$wealth
spatial_gini(x, nb)
```

Description

Update's a spacetime object's number of locations and time periods. A spacetime object's attributes are sticky and will not change if subsetted for example by using `dplyr::filter()` or `dplyr::slice()`. Update the locations and times of a spacetime object.

Usage

```r
spt_update(x, ...)
```

Arguments

- `x`: a spacetime object
- `...`: unused

Value

an object of class spacetime with updated attributes
Description

From an sf object containing points, calculate the standard deviational ellipse.

Usage

std_dev_ellipse(geometry)

Arguments

geometry an sf object. If a polygon, uses `sf::st_point_on_surface()`.

Details

The bulk of this function is derived from the archived CRAN package aspace version 3.2.0.

Value

An sf object with three columns

- sx: major axis radius in CRS units,
- sy: minor axis radius in CRS units,
- theta: degree rotation of the ellipse.

sf object’s geometry is the center mean point.

Examples

```r
# Make a grid to sample from
grd <- sf::st_make_grid(n = c(1, 1), cellsize = c(100, 100), offset = c(0,0))

# sample 100 points
pnts <- sf::st_sample(grd, 100)
std_dev_ellipse(pnts)
```
### std_distance

**Calculate standard distance**

**Description**

The standard distance of a point pattern is a measure of central tendency. Standard distance measures distance away from the mean center of the point pattern similar to standard deviations.

**Usage**

```r
std_distance(geometry)
```

**Arguments**

- `geometry` an sfc object. If a polygon, uses `sf::st_point_on_surface()`.

**Value**

A numeric scalar.

**See Also**

Other point-pattern: `center_mean()`

**Examples**

```r
# Make a grid to sample from
grd <- sf::st_make_grid(n = c(1, 1), cellsize = c(100, 100), offset = c(0,0))

# sample 100 points
pnts <- sf::st_sample(grd, 100)

# plot points
plot(pnts)

# calculate standard distance
std_distance(pnts)
```

---

### st_as_edges

**Convert to an edge lines object**

**Description**

Given geometry and neighbor and weights lists, create an edge list sf object.
Usage

```r
st_as_edges(x, nb, wt)
```

## S3 method for class 'sf'
```r
st_as_edges(x, nb, wt)
```

## S3 method for class 'sfc'
```r
st_as_edges(x, nb, wt)
```

Arguments

- **x**: object of class `sf` or `sfc`.
- **nb**: a neighbor list. If `x` is class `sf`, the unquote named of the column. If `x` is class `sfc`, an object of class `nb` as created from `st_contiguity()`.
- **wt**: optional. A weights list as generated by `st_weights()`. If `x` is class `sf`, the unquote named of the column. If `x` is class `sfc`, the weights list itself.

Details

Creating an edge list creates a column for each `i` position and `j` between an observation and their neighbors. You can recreate these values by expanding the `nb` and `wt` list columns.

```r
code
library(magrittr)
guerry_nb %>%
  tibble::as_tibble() %>%
  dplyr::select(nb, wt) %>%
  dplyr::mutate(i = dplyr::row_number(), .before = 1) %>%
  tidyr::unnest(c(nb, wt))
```

> # A tibble: 420 x 3
> #> i nb wt
> #> <int> <int> <dbl>
> #> 1 1 36 0.25
> #> 2 1 37 0.25
> #> 3 1 67 0.25
> #> 4 1 69 0.25
> #> 5 2 7 0.167
> #> 6 2 49 0.167
> #> 7 2 57 0.167
> #> 8 2 58 0.167
> #> 9 2 73 0.167
> #> 10 2 76 0.167
> #> # ... with 410 more rows
> #> # i Use `print(n = ...)` to see more rows

Value

Returns an `sf` object with edges represented as a `LINESTRING`. 
• from: node index. This is the row position of x.
• to: node index. This is the neighbor value stored in nb.
• i: node index. This is the row position of x.
• j: node index. This is the neighbor value stored in nb.
• wt: the weight value of j stored in wt.

Examples

library(magrittr)
guerry %>%
dplyr::mutate(nb = st_contiguity(geometry),
              wt = st_weights(nb)) %>%
st_as_edges(nb, wt)

Description

Given an sf or sfc object and neighbor and weights lists, create an sfnetwork object.

Usage

st_as_graph(x, nb, wt)

## S3 method for class 'sf'
st_as_graph(x, nb, wt)

## S3 method for class 'sfc'
st_as_graph(x, nb, wt)

Arguments

x object of class sf or sfc.

nb a neighbor list. If x is class sf, the unquote named of the column. If x is class sfc, an object of class nb as created from st_contiguity().

wt optional. A weights list as generated by st_weights(). If x is class sf, the unquote named of the column. If x is class sfc, the weights list itself.

Value

an sfnetwork object

See Also

st_as_nodes() and st_as_edges()
st_as_nodes

Examples

```r
library(magrittr)

guerry_nb %>%
  st_as_graph(nb, wt)
```

---

**st_as_nodes**

*Convert to a node point object*

**Description**

Given geometry and a neighbor list, creates an sf object to be used as nodes in an sfnetwork::sfnetwork(). If the provided geometry is a polygon, sf::st_point_on_surface() will be used to create the node point.

**Usage**

```r
st_as_nodes(x, nb)
```

- **S3 method for class 'sf'**
  ```r
  st_as_nodes(x, nb)
  ```

- **S3 method for class 'sfc'**
  ```r
  st_as_nodes(x, nb)
  ```

**Arguments**

- `x` object of class sf or sfc.
- `nb` a neighbor list. If `x` is class sf, the unquote named of the column. If `x` is class sfc, an object of class nb as created from st_contiguity().

**Details**

`st_as_node`() adds a row `i` based on the attribute "region.id" in the nb object. If the nb object is created with sfdep, then the values will always be row indexes.

**Value**

An object of class sf with POINT geometry.

**Examples**

```r
library(magrittr)

guerry %>%
  dplyr::transmute(nb = st_contiguity(geometry)) %>%
  st_as_nodes(nb)
```
**st_block_nb**  
*Create Block Contiguity for Spatial Regimes*

**Description**

Libpysal write that "block contiguity structures are relevant when defining neighbor relations based on membership in a regime. For example, all counties belonging to the same state could be defined as neighbors, in an analysis of all counties in the US."

Source: libpysal

**Usage**

```
st_block_nb(regime, id = 1:length(regime), diag = FALSE)
```

**Arguments**

- `regime`: a column identifying which spatial regime each element of `id` belongs
- `id`: a column identifying unique observations
- `diag`: default FALSE. If TRUE, includes diagonal element / the self.

**Value**

An object of class `nb`. When `diag = TRUE` the attribute `self.included = TRUE`.

**Examples**

```
id <- guerry$code_dept
regime <- guerry$region
st_block_nb(regime, id)
```

---

**st_cardinalties**  
*Calculate neighbor cardinalities*

**Description**

Identify the cardinality of a neighbor object. Utilizes `spdep::card()` for objects with class `nb`, otherwise returns `lengths(nb)`.

**Usage**

```
st_cardinalties(nb)
```

**Arguments**

- `nb`: A neighbor list object as created by `st_neighbors()`.
**st_complete_nb**

Value

an integer vector with the same length as `nb`.

See Also

Other other: `st_nb_lag_cumul()`, `st_nb_lag()`

Examples

```r
nb <- st_contiguity(sf::st_geometry(guerry))
st_cardinalties(nb)
```

---

**Description**

Create a neighbors list where every element is related to every other element. This creates a complete graph.

**Usage**

```r
st_complete_nb(n_elements, diag = FALSE)
```

**Arguments**

- `n_elements`: the number of observations to create a neighbors list for
- `diag`: default FALSE. If TRUE, includes diagonal element / the self.

**Value**

A neighbors list representing a complete graph.

**Examples**

```r
st_complete_nb(5)
```
**st_contiguity**

Identify polygon neighbors

**Description**

Given an sf geometry of type POLYGON or MULTIPOLYGON identify contiguity based neighbors.

**Usage**

```r
st_contiguity(geometry, queen = TRUE, ...)
```

**Arguments**

- `geometry`: an sf or sfc object.
- `queen`: default TRUE. For more see `?spdep::poly2nb`
- `...`: additional arguments passed to `spdep::poly2nb()`

**Details**

Utilizes `spdep::poly2nb()`

**Value**

a list of class `nb`

**See Also**

Other neighbors: `st_dist_band()`, `st_knn()`

**Examples**

```r
# on basic polygons
geo <- sf::st_geometry(guerry)
st_contiguity(geo)

# in a pipe
library(magrittr)
guerry %>%
dplyr::mutate(nb = st_contiguity(geometry), .before = 1)
```
**st_dist_band**

*Neighbors from a distance band*

**Description**

Creates neighbors based on a distance band. By default, creates a distance band with the maximum distance of k-nearest neighbors where k = 1 (the critical threshold) to ensure that there are no regions that are missing neighbors.

**Usage**

```r
st_dist_band(geometry, lower = 0, upper = critical_threshold(geometry), ...)
```

**Arguments**

- `geometry`: An sf or sfc object.
- `lower`: The lower threshold of the distance band. It is recommended to keep this as 0.
- `upper`: The upper threshold of the distance band. By default is set to a critical threshold using `critical_threshold()` ensuring that each region has a minimum of one neighbor.
- `...`: Passed to `spdep::dnearneigh()`.

**Value**

a list of class `nb`

**See Also**

Other neighbors: `st_contiguity()`, `st_knn()`

**Examples**

```r
geo <- sf::st_geometry(guerry)
st_dist_band(geo, upper = critical_threshold(geo))
```

---

**st_inverse_distance**

*Calculate inverse distance weights*

**Description**

From a neighbor list and sf geometry column, calculate inverse distance weight.

**Usage**

```r
st_inverse_distance(nb, geometry, scale = 100, alpha = 1)
```
st_kernel_weights

Arguments

- **nb** a neighbors list object e.g. created by `st_knn()` or `st_contiguity()`
- **geometry** sf geometry
- **scale** default 100. A value to scale distances by before exponentiating by `alpha`
- **alpha** default 1. Set to 2 for gravity weights.

Details

The inverse distance formula is $w_{ij} = 1/d_{ij}^\alpha$

Value

A list where each element is a numeric vector.

See Also

Other weights: `st_kernel_weights()`, `st_nb_dists()`, `st_weights()`

Examples

```r
geo <- sf::st_geometry(guerry)
b <- st_contiguity(geo)
wts <- st_inverse_distance(nb, geo)
head(wts, 3)
wts <- st_inverse_distance(nb, geo, scale = 10000)
head(wts, 3)
```

---

**Description**

Create a weights list using a kernel function.

**Usage**

```r
st_kernel_weights(
  nb,
  geometry,
  kernel = "uniform",
  threshold = critical_threshold(geometry),
  adaptive = FALSE,
  self_kernel = FALSE
)
```
Arguments

- **nb**
  - an object of class `nb` e.g. created by `st_contiguity()` or `st_knn()`.
- **geometry**
  - the geometry an sf object.
- **kernel**
  - One of "uniform", "gaussian", "triangular", "epanechnikov", or "quartic". See `kernels` for more.
- **threshold**
  - a scaling threshold to be used in calculating
- **adaptive**
  - default FALSE. If TRUE uses the maximum neighbor distance for each region as the threshold. Suppresses the threshold argument.
- **self_kernel**
  - default FALSE. If TRUE applies the kernel function to the observed region.

Details

By default `st_kernel_weight()` utilizes a critical threshold of the maximum neighbor distance using `critical_threshold()`. If desired, the critical threshold can be specified manually. The threshold will be passed to the underlying kernel.

Value

a list where each element is a numeric vector.

See Also

Other weights: `st_inverse_distance()`, `st_nb_dists()`, `st_weights()`

Examples

```r
geometry <- sf::st_geometry(guerry)
nb <- st_contiguity(geometry)
b <- include_self(nb)
res <- st_kernel_weights(nb, geometry)
head(res, 3)
```

---

**st_knn**

*Calculate K-Nearest Neighbors*

Description

Identifies the k nearest neighbors for given point geometry. If polygon geometry is provided, the centroids of the polygon will be used and a warning will be emitted.

Usage

```r
st_knn(geometry, k = 1, symmetric = FALSE, ...)
```
Arguments

- **geometry**: an sf or sfc object.
- **k**: number of nearest neighbours to be returned
- **symmetric**: default FALSE. Whether to force output of neighbours to be symmetric.
- **...**: additional arguments to be passed to `kneigh()`.

Details

This function utilizes `spdep::kneigh()` and `spdep::knn2nb()`.

Value

A list of class `nb`

See Also

Other neighbors: `st_contiguity()`, `st_dist_band()`

Examples

```r
st_knn(sf::st_geometry(guerry), k = 8)
```

---

**st_lag**

*Calculate spatial lag*

Description

Calculates the spatial lag of a numeric variable given a neighbor and weights list.

Usage

```r
st_lag(x, nb, wt, na_ok = FALSE, allow_zero = NULL, ...)
```

Arguments

- **x**: A numeric vector
- **nb**: A neighbor list object as created by `st_neighbors()`.
- **wt**: A weights list as created by `st_weights()`.
- **na_ok**: Default FALSE. If TRUE missing values return a lagged NA.
- **allow_zero**: If TRUE, assigns zero as lagged value to zone without neighbors.
- **...**: See `?spdep::lag.listw` for more.

Value

A numeric vector with same length as `x`
st_nb_apply

See Also

Other stats: local_moran()

Examples

geo <- sf::st_geometry(guerry)
nb <- st_contiguity(geo)
wt <- st_weights(nb)

st_lag(guerry$crime_pers, nb, wt)

---

st_nb_apply Apply a function to neighbors

Description

Sometimes one may want to create custom lag variables or create some other neighborhood level metric that may not be defined yet. This st_nb_apply() enables you to apply a function to each observation's (xi) neighbors (xij).

Usage

st_nb_apply(x, nb, wt, .f, suffix = "dbl", ...)

Arguments

x A vector that will be used for neighbor xij values.
nb A neighbor list object as created by st_neighbors().
wt A weights list as created by st_weights().
.f A function definition. There are three default objects that can be used inside of the function definition:
  • .xij: neighbor values of x for the ith observation. This is simply the subset of x based on the corresponding nb list values for each element.
  • .nb: neighbor positions.
  • .wt: neighbor weights value.

If any of these three function arguments are omitted from .f, dots (…) must be supplied.
suffix The map variant to use. Options are "dbl", "int", "lgl", "chr", "list".
... arguments to pass to .f

Details

The below example calculates the spatial lag using st_nb_apply() and st_lag() to illustrate how we can apply functions to neighbors.

Currently questioning the use case. find_xj() is now exported and may negate the need for this function.
Value

a vector or list of with same length as \( x \).

Examples

```r
library(magrittr)
guerry %>%
dplyr::transmute(
  nb = st_contiguity(geometry),
  wt = st_weights(nb),
  lag_apply = st_nb_apply(
    crime_pers, nb, wt,
    .f = function(.xij, .wt, ...) sum(.xij * .wt)
  ),
  lag = st_lag(crime_pers, nb, wt)
)
```

---

**st_nb_delaunay**  
**Graph based neighbors**

Description

Create graph based neighbors on a set of points.

Usage

```r
st_nb_delaunay(geometry, .id = NULL)
st_nb_gabriel(geometry, .nnmult = 3)
st_nb_relative(geometry, .nnmult = 3)
```

Arguments

- `geometry`: an object of class sfc. If polygons are used, points are generated using `sf::st_point_on_surface()`.
- `.id`: default NULL. Passed as `spdep::tri2nb(x, row.names = .id)` to `spdep`.
- `.nnmult`: default 3. Used for memory scalling. See `spdep::gabrielneigh()` for more.

Details

- `st_nb_delaunay()` uses `spdep::tri2nb()`
- `st_nb_gabriel()` uses `spdep::gabrielneigh()` and `spdep::graph2nb()`
- `st_nb_relative()` uses `spdep::relativeneigh()` and `spdep::graph2nb()`

`st_nb_delaunay()` implements Delaunay triangulation via `spdep` and thus via `deldir`. Delaunay triangulation creates a mesh of triangles that connects all points in a set. It ensures that no point is in the circumcircle of an triangle in the triangulation. As a result, Delaunay triangulation maximizes the minimum angle in each triangle consequently avoiding skinny triangles.
The Gabriel graph is a subgraph of the Delaunay triangulation. Edges are created when the closed disc between two points p, and q, contain no other points besides themselves.

The relative neighborhood graph (RNG) is based on the Delaunay triangulation. It connects two points when there are no other closer points to each of them. The RNG is a subgraph of the Delaunay triangulation.

Note that Delaunay triangulation assumes a plane and thus uses Euclidean distances.

See `spdep::gabrielneigh()` for further descriptions of the graph neighbor implementations.

**Value**

an object of class `nb`

**Examples**

```r
gometry <- sf::st_centroid(sf::st_geometry(guerry))
st_nb_delaunay(geometry)
st_nb_gabriel(geometry)
st_nb_relative(geometry)
```

---

**st_nb_dists**

*Calculate neighbor distances*

**Description**

From an nb list and point geometry, return a list of distances for each observation’s neighbors list.

**Usage**

```r
st_nb_dists(x, nb, longlat = NULL)
```

**Arguments**

- `x` an object of class `sfc`.
- `nb` a neighbor list for example created by `st_contiguity()`
- `longlat` TRUE if point coordinates are longitude-latitude decimal degrees, in which case distances are measured in kilometers. See `?spdep::nbdists()` for more.

**Details**

Utilizes `spdep::nbdists()` for distance calculation.

**Value**

a list where each element is a numeric vector.

**See Also**

Other weights: `st_inverse_distance()`, `st_kernel_weights()`, `st_weights()`
Examples

```r
geo <- sf::st_geometry(guerry)
nb <- st_contiguity(geo)
dists <- st_nb_dists(geo, nb)

head(dists)
```

---

**st_nb_lag**  
**Pure Higher Order Neighbors**

**Description**

Identify higher order neighbors from a neighbor list. `order` must be greater than 1. When `order` equals 2 then the neighbors of the neighbors list is returned and so forth. See Anselin’s slides for an example.

**Usage**

```r
st_nb_lag(nb, order)
```

**Arguments**

- `nb`  
  A neighbor list object as created by `st_contiguity()`.
- `order`  
  The order of neighbors.

**Details**

Utilizes `spdep::nblag()`

**Value**

A list of class `nb`

**See Also**

Other other: `st_cardinalties()`, `st_nb_lag_cumul()`

**Examples**

```r
nb <- st_contiguity(sf::st_geometry(guerry))
st_nb_lag(nb, 3)
```
Description

Creates an encompassing neighbor list of the order specified. For example, if the order is 2 the result contains both 1st and 2nd order neighbors.

Usage

\[ \text{st_nb_lag_cumul}(nb, \text{order}) \]

Arguments

- **nb**: A neighbor list object as created by \text{st_contiguity}().
- **order**: The order of neighbors.

Details

Utilizes \text{spdep::nblag_cumul}()

Value

A list of class \text{nb}

See Also

Other other: \text{st_cardinalties()}, \text{st_nb_lag()}

Examples

```r
nb <- \text{st_contiguity}(\text{sf::st\_geometry(guerry)})
\text{st_nb_lag_cumul}(nb, 3)
```

Description

Calculate polygon spatial weights from a \text{nb} list. See \text{spdep::nb2listw()} for further details.

Usage

\[ \text{st_weights}(\text{nb, style} = \text{"W"}, \text{allow_zero} = \text{NULL, ...}) \]
Arguments

- **nb**: A neighbor list object as created by `st_neighbors()`.
- **style**: Default "W" for row standardized weights. This value can also be "B", "C", "U", "minmax", and "S". See `spdep::nb2listw()` for details.
- **allow.zero**: If TRUE, assigns zero as lagged value to zone without neighbors.
- **...**: additional arguments passed to `spdep::nb2listw()`.

Details

Under the hood, `st_weights()` creates a listw object and then extracts the weights elements from it as the `neighbours` element is already–presumably–already existent in the neighbors list you've already created. listw objects are recreated using `recreate_listw()` when calculating other statistics.

Value

a list where each element is a numeric vector

See Also

Other weights: `st_inverse_distance()`, `st_kernel_weights()`, `st_nb_dists()`

Examples

```r
library(magrittr)
guerry %>%
dplyr::mutate(nb = st_contiguity(geometry),
wt = st_weights(nb),
.before = 1)

# using geometry column directly
nb <- st_contiguity(guerry$geometry)
wt <- st_weights(nb)
w[1:3]
```

---

**szero**  

*Global sum of weights*

Description

Calculate the global sum of weights

Usage

`szero(wt)`
Arguments

wt       a weights list—i.e. created by `st_weights()`

Value

a scalar numeric

Examples

```r
nb <- st_contiguity(guerry)
wt <- st_weights(nb)
szero(wt)
```
wt_as_matrix

Convert neighbor or weights list to matrix

Description

Given a nb list or weights list, convert them to a matrix.

Usage

wt_as_matrix(nb, wt)

nb_as_matrix(nb)

Arguments

nb  
a neighbor list—i.e. as created by st_contiguity().
wt  
a weights list—i.e. as created by st_weights()

Value

Returns a n x n matrix

Examples

# make a grid
g <- sf::st_make_grid(
  cellsize = c(10, 10),
  offset = c(0, 0),
  n = c(2, 2)
)

# create neighbors
nb <- st_contiguity(g)

# cast to matrix
nb_as_matrix(nb)

# create weights
wt <- st_weights(nb)

# cast as matrix
wt_as_matrix(nb, wt)
Index

* datasets
  guerry, 22
* global_c
  global_c, 12
  global_c_perm, 14
  global_c_test, 15
* global_moran
  global_moran, 18
  global_moran_bv, 18
  global_moran_perm, 20
  global_moran_test, 21
  local_moran_bv, 33
* neighbors
  st_contiguity, 52
  st_dist_band, 53
  st_knn, 55
* other
  st_cardinalties, 50
  st_nb_lag, 60
  st_nb_lag_cumul, 61
* point-pattern
  center_mean, 5
  center_median (center_mean), 5
  complete_spacetime_cube, 6
  cond_permute_nb, 7
  critical_threshold, 8
  critical_threshold(), 53
  dplyr::filter(), 44
  dplyr::slice(), 44
  ellipse, 8
  ellipse(), 9
  emerging_hotspot_analysis, 9
  euclidean_median (center_mean), 5
  find_xj, 11
  find_xj(), 57
  global_c, 12, 15
  global_c_perm, 12, 14, 15
  global_c_test, 12, 15, 15
  global_colocation, 13
  global_g_test, 16
  global_jc_perm, 16
  global_jc_test (global_jc_perm), 16
  global_moran, 18, 19–21, 33
  global_moran_bv, 18, 18, 20, 21, 33
  global_moran_perm, 18, 19, 20, 21, 33
  global_moran_test, 18–20, 21, 33
  group_by.spacetime (tidyverse), 63
  guerry, 22
  guerry_nb (guerry), 22
  include_self, 22
  is.spacetime (spacetime), 41
  is.spacetime(), 42
  is_spacetime (spacetime), 41
  is_spacetime(), 42
  is_spacetime_cube, 23
  is_spacetime_cube(), 6
Kendall::MannKendall(), 10
kernels, 55
local_c, 24
local_c_perm, 25
local_c_perm(local_c), 24
local_colocation, 26
local_g, 28
local_g_perm(local_g), 28
local_gstar, 29
local_gstar_perm(local_gstar), 29
local_jc_bv, 30
local_jc_uni, 30
local_moran, 32, 57
local_moran_perm(), 32
local_moran_bv, 18–21, 33
losh, 34
losh_perm(losh), 34
mutate.spacetime(tidyverse), 63
nb_as_matrix(wt_as_matrix), 64
nb_intersect(nb_union), 36
nb_match_test, 35
nb_setdiff(nb_union), 36
nb_union, 36
new_spacetime(spacetime), 41
new_spacetime(), 42
node_get_edge_col(node_get_nbs), 37
node_get_edge_list(node_get_nbs), 37
node_get_nbs, 37
pairwise_colocation, 38
pct_nonzero, 39
pracma::geo_median(), 5
recreate_listw, 40
recreate_listw(), 62
remove_self(include_self), 22
set_col, 40
set_col(), 41
set_nbs(set_col), 40
set_nbs(), 41
set_wts(set_col), 40
set_wts(), 41
sf::st_point_on_surface(), 5, 45, 46, 49
sfnetworks::sfnetwork(), 49
spacetime, 3, 41
spacetime(), 42
spatial_gini, 43
spdep::gabrielneigh(), 58, 59
spdep::geary.mc(), 14
spdep::globalG.test(), 16
spdep::knearneigh(), 56
spdep::knn2nb(), 56
spdep::localC, 25
spdep::localC_perm, 25
spdep::localC_perm(), 25
spdep::localG(), 28, 29
spdep::localG_perm(), 28, 29
spdep::localmoran_perm(), 32
spdep::LOSH, 34
spdep::moran(), 18
spdep::moran.mc(), 15, 20, 21
spdep::nb2listw(), 61, 62
spdep::nblag(), 60
spdep::nblag_cumul(), 61
spdep::poly2nb(), 52
spt_update, 44
st_as_edges, 46
st_as_edges(), 48
st_as_graph, 48
st_as_nodes, 49
st_as_nodes(), 48
st_as_sf(), 4
st_block_nb, 50
st_cardinalties, 50, 60, 61
st_complete_nb, 51
st_contiguity, 52, 53, 56
st_contiguity(), 11, 13, 23, 27, 34, 38, 43, 54, 55, 59
st_dist_band, 52, 53, 56
st_ellipse(ellipse), 8
st_ellipse(), 9
st_inverse_distance, 53, 55, 59, 62
st_kernel_weights, 54, 54, 59, 62
st_kernel_weights(), 26
st_knn, 35, 52, 53, 55
st_knn(), 11, 13, 27, 38, 54, 55
st_lag, 32, 56
st_lag(), 57
st_nb_apply, 57
st_nb_apply(), 57
st_nb_delaunay, 58
st_nb_dists, 54, 55, 59, 62
st_nb_gabriel(st_nb_delaunay), 58
st_nb_lag, 51, 60, 61
INDEX

st_nb_lag_cumul, 51, 60, 61
st_nb_relative (st_nb_delaunay), 58
st_weights, 54, 55, 59, 61
st_weights(), 34, 62
stats::dist(), 35
std_dev_ellipse, 45
std_distance, 6, 46
szero, 62

tally_jc (global_jc_perm), 16
tidyverse, 63

ungroup.spacetime (tidyverse), 63

validate_spacetime (spacetime), 41
validate_spacetime(), 23, 42

wt_as_matrix, 64