Package 'SpaDES.tools'

October 27, 2026

Type Package

Title Additional Tools for Developing Spatially Explicit Discrete Event Simulation (SpaDES) Models

Version 2.2.0

Date 2026-10-27

Description Provides GIS and map utilities, plus additional modeling tools for developing cellular automata, dynamic raster models, and agent based models in 'SpaDES'. Included are various methods for spatial spreading, spatial agents, GIS operations, random map generation, and others. See '?SpaDES.tools' for a categorized overview of these additional tools.

License GPL-3

URL https://spades-tools.predictiveecology.org, https://github.com/PredictiveEcology/SpaDES.tools

BugReports https://github.com/PredictiveEcology/SpaDES.tools/issues

Depends R (>= 4.3)

Imports backports, checkmate (>= 1.8.2), data.table (>= 1.10.4), fpCompare (>= 0.2.1), graphics, methods, parallel, Rcpp (>= 0.12.12), reproducible (>= 2.1.3), stats, terra

Suggests animation, bit (>= 1.1-12), covr, DEoptim (>= 2.2-4), fastmatch, quickPlot (>= 1.0.2), raster (>= 2.5-8), sf, snow, sp (>= 1.2-4), testthat (>= 3.0.0), tools, withr

LinkingTo Rcpp

ByteCompile yes

Config/testthat/edition 3

Encoding UTF-8

Language en-CA

Config/roxygen2/version 8.1.0

Collate 'RcppExports.R' 'heading.R' 'SELES.R'
'distanceFromEachPoint.R' 'environment.R' 'helpers.R'
'initialize.R' 'mapReduce.R' 'mergeRaster.R' 'movement.R'
'neighbourhood.R' 'numerical-comparisons.R' 'probability.R'
'rastFromDF.R' 'release-bullets.R' 'resample.R' 'rings.R'
'spades-tools-deprecated.R' 'spades-tools-package.R'
'splitRaster.R' 'spread-comparison.R' 'spread.R' 'spread2.R'
'spread3.R' 'spreadCpp.R' 'studyArea.R' 'zzz.R'

NeedsCompilation yes

Author Eliot J B McIntire [aut] (ORCID:
<https://orcid.org/0000-0002-6914-8316>),
Alex M Chubaty [aut, cre] (ORCID:
<https://orcid.org/0000-0001-7146-8135>),
Yong Luo [ctb],
Ceres Barros [ctb] (ORCID: <https://orcid.org/0000-0003-4036-977X>),
Steve Cumming [ctb],
Jean Marchal [ctb],
His Majesty the King in Right of Canada, as represented by the Minister
of Natural Resources Canada [cph]

Maintainer Alex M Chubaty <achubaty@for-cast.ca>

Repository CRAN

Date/Publication 2026-10-27 20:50:02 UTC

Contents

SpaDES.tools-package . . . 3
.pointDistance . . . 5
adj . . . 6
agentLocation . . . 9
allInRange01 . . . 10
cir . . . 10
cirSpecialQuick . . . 14
distanceFromEachPoint . . . 15
duplicatedInt . . . 18
dwrpnorm2 . . . 18
fastCrop . . . 19
gaussMap . . . 20
heading . . . 21
initiateAgents . . . 22
inRange . . . 23
mergeRaster . . . 24
middlePixel . . . 27
move . . . 28
neutralLandscapeMap . . . 30
numAgents . . . 31
patchSize . . . 32

probInit 32
randomPolygons 33
randomStudyArea 35
rasterizeReduced 36
rastFromDF 37
rings 39
runifC 41
specificNumPerPatch 42
spokes 43
spread 46
spread2 56
spread3 65
spreadCpp 70
testEquivalentMetadata 74
transitions 75
wrapTorus 75

Index **78**

`SpaDES.tools-package` *Categorized overview of the* `SpaDES.tools` *package*

Description

1 Spatial spreading/distances methods

Spatial contagion is a key phenomenon for spatially explicit simulation models. Contagion can be modelled using discrete approaches or continuous approaches. Several functions assist with these:

`adj()`	Fast neighbour finding, delegating to `terra::adjacent()`, with a faster torus option
`cir()`	Identify pixels in a circle around a `SpatVector` object
`directionFromEachPoint()`	Fast calculation of direction and distance surfaces
`distanceFromEachPoint()`	Fast calculation of distance surfaces
`rings()`	Identify rings around focal cells (e.g., buffers and donuts)
`spokes()`	Identify pixels along rays radiating out from focal cells
`spread()`	Contagious cellular automata
`wrapTorus()`	Create a torus from a grid

2 Spatial agent methods

Agents have several methods and functions specific to them:

`crw()`	Simple correlated random walk function
`heading()`	Determines the heading between points
`quickPlot::makeLines()`	Makes `SpatialLines` object for, e.g., drawing arrows
`move()`	A meta function that can currently only take "crw"
`specificNumPerPatch()`	Initiate a specific number of agents per patch

3 GIS operations

In addition to the GIS operations available in R (chiefly from `terra` and `sf`), we provide the following GIS-related functions:

`quickPlot::equalExtent()`	Assess whether a list of extents are all equal

4 Map-reduce - type operations

These functions convert between reduced and mapped representations of the same data. This allows compact representation of, e.g., rasters that have many individual pixels that share identical information.

`rasterizeReduced()`	Convert reduced representation to full raster

5 Random Map Generation

It is often useful to build dummy maps with which to build simulation models before all data are available. These dummy maps can later be replaced with actual data maps.

`randomPolygons()`	Creates a random polygon with specified number of classes.

See `neutralLandscapeMap()` for tools to generate random landscapes (rasters).

6 SELES-type approach to simulation

These functions are essentially skeletons and are not fully implemented. They are intended to make translations from SELES. You must know how to use SELES for these to be useful:

`agentLocation()`	Agent location
`initiateAgents()`	Initiate agents into a `SpatVector`
`numAgents()`	Number of agents
`probInit()`	Probability of initiating an agent or event

7 Package options

SpaDES packages use the following `options()` to configure behaviour:

- `spades.lowMemory`: If true, some functions will use more memory efficient (but slower) algorithms. Default `FALSE`.
- `spades.useSpreadCpp`: If `TRUE`, `spread()` and `spread2()` hand their calls to the faster `spreadCpp()` where it supports the arguments used. Results differ for the same seed. Default `FALSE`.

Author(s)

Maintainer: Alex M Chubaty <achubaty@for-cast.ca> (ORCID)

Authors:

- Alex M Chubaty <achubaty@for-cast.ca> (ORCID)
- Eliot J B McIntire <eliot.mcintire@nrcan-rncan.gc.ca> (ORCID)

Other contributors:

- Yong Luo <Yong.Luo@gov.bc.ca> [contributor]
- Ceres Barros <ceres.barros@ubc.ca> (ORCID) [contributor]
- Steve Cumming <Steve.Cumming@sbf.ulaval.ca> [contributor]
- Jean Marchal <jean.d.marchal@gmail.com> [contributor]
- His Majesty the King in Right of Canada, as represented by the Minister of Natural Resources Canada [copyright holder]

See Also

Useful links:

- https://spades-tools.predictiveecology.org
- https://github.com/PredictiveEcology/SpaDES.tools
- Report bugs at https://github.com/PredictiveEcology/SpaDES.tools/issues

---

`.pointDistance` *Alternative point distance (and direction) calculations*

Description

These have been written with speed in mind.

Usage

```
.pointDistance(
  from,
  to,
  angles = NA,
  maxDistance = NA_real_,
  otherFromCols = FALSE
)
```

Arguments

from	Numeric matrix with 2 or 3 or more columns. They must include x and y, representing x and y coordinates of "from" cell. If there is a column named "id", it will be "id" from to, i.e,. specific pair distances. All other columns will be included in the return value of the function.
to	Numeric matrix with 2 or 3 columns (or optionally more, all of which will be returned), x and y, representing x and y coordinates of "to" cells, and optional "id" which will be matched with "id" from from. Default is all cells.
angles	Logical. If TRUE, then the function will return angles in radians, as well as distances.
maxDistance	Numeric in units of number of cells. The algorithm will build the whole surface (from from to to), but will remove all distances that are above this distance. Using this will keep memory use down.
otherFromCols	other columns to use as 'from'

Value

A matrix with one row per from-to pair, carrying the columns of from plus "dists", the distance between from and to. angles = TRUE adds an "angles" column, otherFromCols adds the named columns of from, and a non-NA maxDistance adds "keptIndex", the index of each retained pair in the unfiltered set.

---

adj *Fast* adjacent *function, and Just In Time compiled version*

Description

Faster function for determining the cells of the 4, 8 or bishop neighbours of the cells. This is a hybrid function that uses matrix for small numbers of loci (<1e4) and data.table for larger numbers of loci

Usage

```
adj(
  x = NULL,
  cells,
  directions = 8,
  sort = FALSE,
  pairs = TRUE,
  include = FALSE,
  target = NULL,
  numCol = NULL,
  numCell = NULL,
  match.adjacent = FALSE,
  cutoff.for.data.table = 2000,
  torus = FALSE,
  id = NULL,
  numNeighs = NULL,
  returnDT = FALSE
)
```

Arguments

x	SpatRaster object for which adjacency will be calculated.
cells	vector of cell numbers for which adjacent cells should be found. Cell numbers start with 1 in the upper-left corner and increase from left to right and from top to bottom.
directions	the number of directions in which cells should be connected: 4 (rook's case), 8 (queen's case), or "bishop" to connect cells with one-cell diagonal moves. Or a neighbourhood matrix (see Details).
sort	logical. Whether the outputs should be sorted or not, using cell ids of the from cells (and to cells, if match.adjacent is TRUE).
pairs	logical. If TRUE, a matrix of pairs of adjacent cells is returned. If FALSE, a vector of cells adjacent to cells is returned
include	logical. Should the focal cells be included in the result?
target	a vector of cells that can be spread to. This is the inverse of a mask.
numCol	numeric indicating number of columns in the raster. Using this with numCell is a bit faster execution time.
numCell	numeric indicating number of cells in the raster. Using this with numCol is a bit faster execution time.
match.adjacent	logical. Should the returned object be the same as raster::adjacent. Default FALSE, which is faster.
cutoff.for.data.table	numeric. If the number of cells is above this value, the function uses data.table internally, which is faster with large numbers of cells. Default is 2000, which appears to be the turning point where data.table becomes faster. This selects the internal strategy only; it does not affect the class of the returned object (see returnDT).

torus	Logical. Should the spread event wrap around to the other side of the raster? Default is FALSE.
id	numeric If not NULL (default), then function will return "id" column.
numNeighs	A numeric scalar, indicating how many neighbours to return. Must be less than or equal to directions; which neighbours are random with equal probabilities.
returnDT	A logical. If TRUE, the result is returned as a data.table, whichever internal strategy was used to build it. Up to version 2.1.2 this was honoured only when length(cells) was greater than cutoff.for.data.table; below that the argument was ignored and a matrix was returned. The first call per session that lands in that window warns.

Details

Between 4x (large number loci) to 200x (small number loci) speed gains over adjacent in raster package. There is some extra speed gain if NumCol and NumCells are passed rather than a raster. Efficiency gains come from:

1. use data.table internally
    - no need to remove NAs because wrapped or outside points are just removed directly with data.table
    - use data.table to sort and fast select (though not fastest possible)
2. don't make intermediate objects; just put calculation into return statement

The steps used in the algorithm are:

1. Calculate indices of neighbouring cells
2. Remove "to" cells that are
    - < 1 or > numCells (i.e., they are above or below raster), using a single modulo calculation
    - where the modulo of "to" cells is equal to 1 if "from" cells are 0 (wrapped right to left)
    - or where the modulo of the "to" cells is equal to 0 if "from" cells are 1 (wrapped left to right)

Value

If returnDT is TRUE, a data.table; otherwise a matrix, except for pairs = FALSE combined with match.adjacent = TRUE, which returns a bare vector of the unique to cells. The columns will be one or more of id, from, to: to always; from if pairs is TRUE; id if id was supplied. The variable output is done to minimize coercion to maintain speed.

Author(s)

Eliot McIntire

See Also

terra::adjacent()

Examples

```
library(terra)

origDTThreads <- data.table::setDTthreads(2L)
origNcpus <- options(Ncpus = 2L)

a <- rast(ext(0, 1000, 0, 1000), resolution = 1)
sam <- sample(1:ncell(a), 1e4)
numCol <- ncol(a)
numCell <- ncell(a)
adj.new <- adj(numCol = numCol, numCell = numCell, cells = sam, directions = 8)
adj.new <- adj(numCol = numCol, numCell = numCell, cells = sam, directions = 8,
               include = TRUE)

# clean up
data.table::setDTthreads(origDTThreads)
options(Ncpus = origNcpus)
```

---

agentLocation SELES - *Agent Location at initiation*

Description

Sets the the location of the initiating agents. NOT YET FULLY IMPLEMENTED.

A SELES-like function to maintain conceptual backwards compatibility with that simulation tool. This is intended to ease transitions from SELES.

You must know how to use SELES for these to be useful.

Usage

```
agentLocation(map)
```

Arguments

map	A SpatialPoints*, SpatialPolygons*, or Raster* object.

Value

Object of same class as provided as input. If a Raster*, then zeros are converted to NA.

Author(s)

Eliot McIntire

allInRange01 *Are all values in* [0, 1]*, ignoring* NA*?*

Description

The question spread() asks of spreadProb before it starts. Expressed in R as all(inRange(na.omit(x))) it costs three passes and two copies of the whole vector – na.omit() allocates one, the comparison another – which is work proportional to the size of the landscape on every call, however little of it burns. This is one pass, no allocation, and it stops at the first value that fails.

NA and NaN are ignored, so an all-NA vector and a zero-length vector are both TRUE: there is nothing in them that is out of range. That matches all(inRange(na.omit(x))), which is all(logical(0)).

Usage

```
allInRange01(x)
```

Arguments

x	Numeric vector.

Value

A single logical.

cir *Identify pixels in a circle or ring (doughnut) around an object.*

Description

Identify the pixels and coordinates that are at a (set of) buffer distance(s) of the objects passed into coords. This is similar to sf::st_buffer but much faster and without the georeferencing information. In other words, it can be used for similar problems, but where speed is important. This code is substantially adapted from PlotRegionHighlighter::createCircle.

Usage

```
cir(
  landscape,
  coords,
  loci,
  maxRadius = ncol(landscape)/4,
  minRadius = maxRadius,
  allowOverlap = TRUE,
  allowDuplicates = FALSE,
  includeBehavior = "includePixels",
```

```
  returnDistances = FALSE,
  angles = NA_real_,
  returnAngles = FALSE,
  returnIndices = TRUE,
  closest = FALSE,
  simplify = TRUE
)
```

Arguments

landscape	Raster on which the circles are built.
coords	Either a matrix with 2 (or 3) columns, x and y (and id), representing the coordinates (and an associated id, like cell index), or a SpatialPoints* object around which to make circles. Must be same coordinate system as the landscape argument. Default is missing, meaning it uses the default to loci.
loci	Numeric. An alternative to coords. These are the indices on landscape to initiate this function (see coords). Default is one point in centre of landscape.
maxRadius	Numeric vector of length 1 or same length as coords
minRadius	Numeric vector of length 1 or same length as coords. Default is maxRadius, meaning return all cells that are touched by the narrow ring at that exact radius. If smaller than maxRadius, then this will create a buffer or doughnut or ring.
allowOverlap	Logical. Should duplicates across id be removed or kept. Default TRUE.
allowDuplicates	Logical. Should duplicates within id be removed or kept. Default FALSE. This is useful if the actual x, y coordinates are desired, rather than the cell indices. This will increase the size of the returned object.
includeBehavior	Character string. Currently accepts only "includePixels", the default, and "excludePixels". See details.
returnDistances	Logical. If TRUE, then a column will be added to the returned data.table that reports the distance from coords to every point that was in the circle/doughnut surrounding coords. Default FALSE, which is faster.
angles	Numeric. Optional vector of angles, in radians, to use. This will create "spokes" outward from coords. Default is NA, meaning, use internally derived angles that will "fill" the circle.
returnAngles	Logical. If TRUE, then a column will be added to the returned data.table that reports the angle from coords to every point that was in the circle/doughnut surrounding coords. Default FALSE.
returnIndices	Logical or numeric. If 1 or TRUE, will return a data.table with indices and values of successful spread events. If 2, it will simply return a vector of pixel indices of all cells that were touched. This will be the fastest option. If FALSE, then it will return a raster with values. See Details.
closest	Logical. When determining non-overlapping circles, should the function give preference to the closest loci or the first one (much faster). Default is FALSE, meaning the faster, though maybe not desired behaviour.

simplify — logical. If TRUE, then all duplicate pixels are removed. This means that some x, y combinations will disappear.

Details

This function identifies all the pixels as defined by a donut with inner radius minRadius and outer radius of maxRadius. The includeBehavior defines whether the cells that intersect the radii but whose centres are not inside the donut are included includePixels or not excludePixels in the returned pixels identified. If this is excludePixels, and if a minRadius and maxRadius are equal, this will return no pixels.

Value

A matrix with 4 columns, id, indices, x, y. The x and y indicate the exact coordinates of the indices (i.e., cell number) of the landscape associated with the ring or circle being identified by this function.

See Also

rings() which uses spread internally. cir tends to be faster when there are few starting points, rings tends to be faster when there are many starting points. cir scales with maxRadius^2 and coords. Another difference between the two functions is that rings takes the centre of the pixel as the centre of a circle, whereas cir takes the exact coordinates. See example. For the specific case of creating distance surfaces from specific points, see distanceFromEachPoint(), which is often faster. For the more general GIS buffering, see sf::st_buffer.

Examples

```
library(data.table)
library(terra)

origDTThreads <- data.table::setDTthreads(2L)
origNcpus <- options(Ncpus = 2L)
set.seed(1462)

# circle centred
ras <- rast(ext(0, 15, 0, 15), res = 1, val = 0)
middleCircle <- cir(ras)
ras[middleCircle[, "indices"]] <- 1
circlePoints <- vect(middleCircle[, c("x", "y")])
if (interactive()) {
  # clearPlot()
  terra::plot(ras)
  terra::plot(circlePoints, add = TRUE)
}

# circles non centred
ras <- randomPolygons(ras, numTypes = 4)
n <- 2
agent <- vect(cbind(x = stats::runif(n, xmin(ras), xmax(ras)),
                    y = stats::runif(n, xmin(ras), xmax(ras))))
```

```
cirs <- cir(ras, agent, maxRadius = 15, simplify = TRUE) ## TODO: empty with some seeds! e.g. 1642
cirsSP <- vect(cirs[, c("x", "y")]) ## TODO: error with some seeds! e.g. 1642
cirsRas <- rast(ras)
cirsRas[] <- 0
cirsRas[cirs[, "indices"]] <- 1

if (interactive()) {
  terra::plot(ras)
  terra::plot(cirsRas, add = TRUE, col = c("transparent", "#00000055"))
  terra::plot(agent, add = TRUE)
  terra::plot(cirsSP, add = TRUE)
}

# Example comparing rings and cir
hab <- rast(system.file("extdata", "hab1.tif", package = "SpaDES.tools"))
radius <- 4
n <- 2
coords <- vect(cbind(x = stats::runif(n, xmin(hab), xmax(hab)),
                     y = stats::runif(n, xmin(hab), xmax(hab))))

# cirs
cirs <- cir(hab, coords, maxRadius = rep(radius, length(coords)), simplify = TRUE)

ras1 <- rast(hab)
ras1[] <- 0
ras1[cirs[, "indices"]] <- cirs[, "id"]
if (interactive()) {
  terra::plot(ras1)
}

# rings
loci <- cellFromXY(hab, crds(coords))
cirs2 <- rings(hab, loci, maxRadius = radius, minRadius = radius - 1, returnIndices = TRUE)

ras2 <- rast(hab)
ras2[] <- 0
ras2[cirs2$indices] <- cirs2$id
if (interactive()) {
  terra::plot(c(ras1, ras2))
}

hab <- rast(system.file("extdata", "hab2.tif", package = "SpaDES.tools"))
cirs <- cir(hab, coords, maxRadius = 44, minRadius = 0)
ras1 <- rast(hab)
ras1[] <- 0
cirsOverlap <- data.table::data.table(cirs)[, list(sumIDs = sum(id)), by = indices]
ras1[cirsOverlap$indices] <- cirsOverlap$sumIDs
if (interactive()) {
  terra::plot(ras1)
}

# Provide a specific set of angles
```

```
ras <- rast(ext(0, 330, 0, 330), res = 1)
ras[] <- 0
n <- 2
coords <- cbind(x = stats::runif(n, xmin(ras), xmax(ras)),
                y = stats::runif(n, xmin(ras), xmax(ras)))
circ <- cir(ras, coords, angles = seq(0, 2 * pi, length.out = 21),
            maxRadius = 200, minRadius = 0, returnIndices = FALSE,
            allowOverlap = TRUE, returnAngles = TRUE)

# clean up
data.table::setDTthreads(origDTThreads)
options(Ncpus = origNcpus)
```

`cirSpecialQuick` *This is a very fast version of* `cir` *with* `allowOverlap = TRUE`, `allowDuplicates = FALSE`, `returnIndices = TRUE`, `returnDistances = TRUE`, *and* `includeBehavior = "excludePixels"`. *It is used inside* `spread2`, *when asymmetry is active. The basic algorithm is to run* `cir` *just once, then add to the x,y coordinates of every locus.*

Description

This is a very fast version of `cir` with `allowOverlap = TRUE`, `allowDuplicates = FALSE`, `returnIndices = TRUE`, `returnDistances = TRUE`, and `includeBehavior = "excludePixels"`. It is used inside `spread2`, when asymmetry is active. The basic algorithm is to run `cir` just once, then add to the x,y coordinates of every locus.

Usage

```
.cirSpecialQuick(landscape, loci, maxRadius, minRadius)
```

Arguments

`landscape`	Raster on which the circles are built.
`loci`	Numeric. An alternative to `coords`. These are the indices on `landscape` to initiate this function (see `coords`). Default is one point in centre of `landscape`.
`maxRadius`	Numeric vector of length 1 or same length as `coords`
`minRadius`	Numeric vector of length 1 or same length as `coords`. Default is `maxRadius`, meaning return all cells that are touched by the narrow ring at that exact radius. If smaller than `maxRadius`, then this will create a buffer or doughnut or ring.

distanceFromEachPoint *Calculate distances and directions between many points and many grid cells*

Description

This is a modification of terra::distance() for the case of many points. This version can often be faster for a single point because it does not return a RasterLayer. This is different than terra::distance() because it does not take the minimum distance from the set of points to all cells. Rather this returns the every pair-wise point distance. As a result, this can be used for doing inverse distance weightings, seed rain, cumulative effects of distance-based processes etc. If memory limitation is an issue, maxDistance will keep memory use down, but with the consequences that there will be a maximum distance returned. This function has the potential to use a lot of memory if there are a lot of from and to points.

Usage

```
distanceFromEachPoint(
  from,
  to = NULL,
  landscape,
  angles = NA_real_,
  maxDistance = NA_real_,
  cumulativeFn = NULL,
  distFn = function(dist) 1/(1 + dist),
  cl,
  ...
)
```

Arguments

from	Numeric matrix with 2 or 3 or more columns. They must include x and y, representing x and y coordinates of "from" cell. If there is a column named "id", it will be "id" from to, i.e,. specific pair distances. All other columns will be included in the return value of the function.
to	Numeric matrix with 2 or 3 columns (or optionally more, all of which will be returned), x and y, representing x and y coordinates of "to" cells, and optional "id" which will be matched with "id" from from. Default is all cells.
landscape	RasterLayer. optional. This is only used if to is NULL, in which case all cells are considered to.
angles	Logical. If TRUE, then the function will return angles in radians, as well as distances.
maxDistance	Numeric in units of number of cells. The algorithm will build the whole surface (from from to to), but will remove all distances that are above this distance. Using this will keep memory use down.

cumulativeFn	A function that can be used to incrementally accumulate values in each to location, as the function iterates through each from. See Details.
distFn	A function. This can be a function of landscape, fromCell (single integer value of a from pixel), toCells (integer vector value of all the to pixel indices), and dist. If cumulativeFn is supplied, this will be used to convert the distances to some other set of units that will be accumulated by the cumulativeFn. See Details and examples.
cl	A cluster object. Optional. This would generally be created using parallel::makeCluster() or equivalent. This is an alternative way, instead of beginCluster(), to use parallelism for this function, allowing for more control over cluster use.
...	Any additional objects needed for distFn.

Details

This function is cluster aware if the raster package is available. If there is a cluster running, it will use it. To start a cluster use raster::beginCluster(), with N being the number of cores to use. See examples in SpaDES.core::experiment.

If the user requires an id (indicating the from cell for each to cell) to be returned with the function, the user must add an identifier to the from matrix, such as "id". Otherwise, the function will only return the coordinates and distances.

distanceFromEachPoint calls .pointDistance, which is not intended to be called directly by the user.

This function has the potential to return a very large object, as it is doing pairwise distances (and optionally directions) between from and to. If there are memory limitations because there are many from and many to points, then cumulativeFn and distFn can be used. These two functions together will be used iteratively through the from points. The distFn should be a transformation of distances to be used by the cumulativeFn function. For example, if distFn is 1 / (1+x), the default, and cumulativeFn is +, then it will do a sum of inverse distance weights. See examples.

Value

A sorted matrix on id with same number of rows as to, but with one extra column, "dists", indicating the distance between from and to.

See Also

rings(), cir(), terra::distance(), which can all be made to do the same thing, under specific combinations of arguments. But each has different primary use cases. Each is also faster under different conditions. For instance, if maxDistance is relatively small compared to the number of cells in the landscape, then cir() will likely be faster. If a minimum distance from all cells in the landscape to any cell in from, then distanceFromPoints will be fastest. This function scales best when there are many to points or all cells are used to = NULL (which is default).

Examples

```
library(terra)

origDTThreads <- data.table::setDTthreads(2L)
```

```
origNcpus <- options(Ncpus = 2L)

n <- 2
distRas <- rast(ext(0, 40, 0, 40), res = 1)
coords <- cbind(x = round(runif(n, xmin(distRas), xmax(distRas))) + 0.5,
                y = round(runif(n, xmin(distRas), xmax(distRas))) + 0.5)

# inverse distance weights
dists1 <- distanceFromEachPoint(coords, landscape = distRas)
indices <- cellFromXY(distRas, dists1[, c("x", "y")])
invDist <- tapply(dists1[, "dists"], indices, function(x) sum(1 / (1 + x))) # idw function
distRas[] <- as.vector(invDist)
if (interactive()) {
  # clearPlot()
  terra::plot(distRas)
}

# With iterative summing via cumulativeFn to keep memory use low, with same result
dists1 <- distanceFromEachPoint(coords[, c("x", "y"), drop = FALSE],
                                landscape = distRas, cumulativeFn = `+`)
idwRaster <- rast(distRas)
idwRaster[] <- dists1[, "dists"]
if (interactive()) terra::plot(idwRaster)

all(idwRaster[] == distRas[]) # TRUE

# A more complex example of cumulative inverse distance sums, weighted by the value
#  of the origin cell
ras <- rast(ext(0, 34, 0, 34), res = 1, val = 0)
rp <- randomPolygons(ras, numTypes = 10) ^ 2
n <- 15
cells <- sample(ncell(ras), n)
coords <- xyFromCell(ras, cells)
distFn <- function(landscape, fromCell, dist) landscape[fromCell] / (1 + dist)

#beginCluster(3) # can do parallel
dists1 <- distanceFromEachPoint(coords[, c("x", "y"), drop = FALSE],
                                landscape = rp, distFn = distFn, cumulativeFn = `+`)
#endCluster() # if beginCluster was run

idwRaster <- rast(ras)
idwRaster[] <- dists1[, "dists"]
if (interactive()) {
  # clearPlot()
  terra::plot(rp)
  sp1 <- vect(coords)
  terra::plot(sp1, add = TRUE)
  terra::plot(idwRaster)
  terra::plot(sp1, add = TRUE)
}

# On linux; can use numeric passed to cl; will use mclapply with mc.cores = cl
if (identical(Sys.info()["sysname"], "Linux")) {
```

```
  dists1 <- distanceFromEachPoint(coords[, c("x", "y"), drop = FALSE],
                                  landscape = rp, distFn = distFn,
                                  cumulativeFn = `+`, cl = 2)
}

# clean up
data.table::setDTthreads(origDTThreads)
options(Ncpus = origNcpus)
```

---

duplicatedInt *Rcpp duplicated on integers using Rcpp Sugar*

Description

`.duplicatedInt` does same as `duplicated` in R, but only on integers, and faster. It uses Rcpp sugar

Usage

```
duplicatedInt(x)
```

Arguments

`x`	Integer Vector

Value

A logical vector, as per `duplicated`

---

dwrpnorm2 *Vectorized wrapped normal density function*

Description

This is a modified version of `dwrpnorm` in the `CircStats` package to allow for multiple angles at once (i.e., vectorized on `theta` and `mu`).

Usage

```
dwrpnorm2(theta, mu, rho, sd = 1, acc = 1e-05, tol = acc)
```

Arguments

theta	value at which to evaluate the density function, measured in radians.
mu	mean direction of distribution, measured in radians.
rho	mean resultant length of distribution.
sd	different way of select rho, see details below.
acc	parameter defining the accuracy of the estimation of the density. Terms are added to the infinite summation that defines the density function until successive estimates are within acc of each other.
tol	same as acc.

Value

A numeric vector of densities, of length `max(length(theta), length(mu), length(rho))`; `theta` and `mu` are recycled to that length.

Author(s)

Eliot McIntire

Examples

```
# Values for which to evaluate density
theta <- c(1:500) * 2 * pi / 500
# Compute wrapped normal density function
density <- c(1:500)
for(i in 1:500) {
  density[i] <- dwrpnorm2(theta[i], pi, .75)
}

if (interactive()) {
  plot(theta, density)
}

# Approximate area under density curve
sum(density * 2 * pi / 500)
```

---

fastCrop fastCrop *is deprecated.*

Description

`fastCrop` is deprecated.

Usage

```
fastCrop(x, y, ...)
```

Arguments

x	Raster to crop
y	Raster to crop with
...	other

Value

None. Called for its side effect of signalling the deprecation condition; use `terra::crop()` instead.

See Also

`velox::VeloxRaster_crop`

---

gaussMap *Produce a* raster *of a random Gaussian process.*

Description

Defunct.

Usage

```
gaussMap(
  x,
  scale = 10,
  var = 1,
  speedup = 1,
  method = "RMexp",
  alpha = 1,
  inMemory = FALSE,
  ...
)
```

Arguments

x	A spatial object (e.g., a RasterLayer).
scale	The spatial scale in map units of the Gaussian pattern.
var	Spatial variance.
speedup	An numeric value indicating how much faster than 'normal' to generate maps. It may be necessary to give a value larger than 1 for large maps. Default is 1.
method	The type of model used to produce the Gaussian pattern. Should be one of "RMgauss" (Gaussian covariance model), "RMstable" (the stable powered exponential model), or the default, "RMexp" (exponential covariance model).
alpha	A required parameter of the "RMstable" model. Should be in the interval [0,2] to provide a valid covariance function. Default is 1.
inMemory	Should the RasterLayer be forced to be in memory? Default FALSE.
...	Additional arguments to raster.

Value

A raster map with same extent as x, with a Gaussian random pattern.

---

heading *Heading between spatial points.*

Description

Determines the heading between spatial points.

Usage

```
heading(from, to)
```

Arguments

from	The starting position; an object of class SpatVector.
to	The ending position; an object of class SpatVector.

Value

The heading between the points, in degrees.

Author(s)

Eliot McIntire

Examples

```
library(terra)

origDTThreads <- data.table::setDTthreads(2L)
origNcpus <- options(Ncpus = 2L)

N <- 10L                       # number of agents
x1 <- stats::runif(N, -50, 50) # previous X location
y1 <- stats::runif(N, -50, 50) # previous Y location
x0 <- stats::rnorm(N, x1, 5)   # current X location
y0 <- stats::rnorm(N, y1, 5)   # current Y location

# using SpatVector
prev <- terra::vect(cbind(x = x1, y = y1))
curr <- terra::vect(cbind(x = x0, y = y0))
heading(prev, curr)

# using matrix
prev <- matrix(c(x1, y1), ncol = 2, dimnames = list(NULL, c("x","y")))
curr <- matrix(c(x0, y0), ncol = 2, dimnames = list(NULL, c("x","y")))
heading(prev, curr)
```

```
#using both
prev <- terra::vect(cbind(x = x1, y = y1))
curr <- matrix(c(x0, y0), ncol = 2, dimnames = list(NULL, c("x","y")))
heading(prev, curr)

prev <- matrix(c(x1, y1), ncol = 2, dimnames = list(NULL, c("x","y")))
curr <- terra::vect(cbind(x = x0, y = y0))
heading(prev, curr)

# clean up
data.table::setDTthreads(origDTThreads)
options(Ncpus = origNcpus)
```

`initiateAgents` SELES - *Initiate agents*

Description

Sets the the number of agents to initiate. THIS IS NOT FULLY IMPLEMENTED.

A SELES-like function to maintain conceptual backwards compatibility with that simulation tool. This is intended to ease transitions from SELES.

You must know how to use SELES for these to be useful.

Usage

```
initiateAgents(map, numAgents, probInit, asSpatialPoints = TRUE, indices)
```

Arguments

`map`	`RasterLayer` with extent and resolution of desired return object
`numAgents`	numeric resulting from a call to `numAgents()`
`probInit`	a `Raster` resulting from a `probInit()` call
`asSpatialPoints`	logical or `"sf"`. Should returned object be `RasterLayer` or `SpatVector` default (or legacy `TRUE` `SpatialPointsDataFrame`)
`indices`	numeric. Indices of where agents should start

Value

A `SpatialPointsDataFrame`, with each row representing an individual agent

Author(s)

Eliot McIntire

Examples

```
if (require("sf", quietly = TRUE)) {
  library(data.table)
  library(terra)

  origDTThreads <- data.table::setDTthreads(2L)
  origNcpus <- options(Ncpus = 2L)

  map <- rast(system.file("extdata", "map.tif", package = "SpaDES.tools"))
  names(map) <- "layer"
  pr <- probInit(map, p = (map[] / terra::minmax(map)[2])^2)
  agents <- initiateAgents(map, 100, pr, asSpatialPoints = "sf")
  if (interactive()) {
    terra::plot(map)
    terra::plot(agents, add = TRUE)
  }
  # Test that they are indeed selecting according to probabilities in pr
  dt1 <- data.table(table(round(extract(map, agents), 0)[, "layer"]))
  setnames(dt1, old = "N", new = "count")
  dt2 <- data.table(table(round(map[], 0)))
  setnames(dt2, old = "N", new = "available")
  dt <- dt1[dt2, on = "V1"]  # join the counts and available data.tables
  setnames(dt, old = "V1", new = "mapValue")
  dt[, selection := count / available]
  dt[is.na(selection), selection := 0]
  if (interactive()) {
    with(dt, plot(mapValue, selection))
  }
  #'
  # Note, can also produce a Raster representing agents,
  # then the number of points produced can't be more than
  # the number of pixels:
  agentsRas <- initiateAgents(map, 30, pr, asSpatialPoints = FALSE)
  if (interactive()) {
    terra::plot(agentsRas)
  }
  #'
  # clean up
  data.table::setDTthreads(origDTThreads)
  options(Ncpus = origNcpus)
}
```

---

inRange *Test whether a number lies within range* [a,b]

Description

Default values of a=0; b=1 allow for quick test if x is a probability.

Usage

```
inRange(x, a = 0, b = 1)
```

Arguments

x	values to be tested
a	lower bound (default 0)
b	upper bound (default 1)

Value

Logical vectors. NA values in x are retained.

Author(s)

Alex Chubaty

Examples

```
set.seed(100)
x <- stats::rnorm(4) # -0.50219235  0.13153117 -0.07891709  0.88678481
inRange(x, 0, 1)
```

---

mergeRaster *Split and re-merge* RasterLayer*(s)*

Description

splitRaster divides up a raster into an arbitrary number of pieces (tiles). Split rasters can be recombined using do.call(merge, y) or mergeRaster(y), where y <- splitRaster(x).

Usage

```
mergeRaster(x, fun = NULL)

## S4 method for signature 'list'
mergeRaster(x, fun = NULL)

splitRaster(
  r,
  nx = 1,
  ny = 1,
  buffer = c(0, 0),
  path = NA,
  cl,
  rType = "FLT4S",
  fExt = ".tif"
)
```

Arguments

x	A list of split raster tiles (i.e., from splitRaster).
fun	Function (e.g. mean, min, or max that accepts a na.rm argument. The default is mean.
r	The raster to be split.
nx	The number of tiles to make along the x-axis.
ny	The number of tiles to make along the y-axis.
buffer	Numeric vector of length 2 giving the size of the buffer along the x and y axes. If values greater than or equal to 1 are used, this is interpreted as the number of pixels (cells) to use as a buffer. Values between 0 and 1 are interpreted as proportions of the number of pixels in each tile (rounded up to an integer value). Default is c(0, 0), which means no buffer.
path	Character specifying the directory to which the split tiles will be saved. If missing, the function will write to memory.
cl	A cluster object. Optional. This would generally be created using parallel::makeCluster() or equivalent. This is an alternative way, instead of beginCluster(), to use parallelism for this function, allowing for more control over cluster use.
rType	Data type of the split rasters. Defaults to FLT4S.
fExt	file extension (e.g., ".grd" or ".tif") specifying the file format.

Details

mergeRaster differs from merge in how overlapping tile regions are handled: merge retains the values of the first raster in the list. This has the consequence of retaining the values from the buffered region in the first tile in place of the values from the neighbouring tile. On the other hand, mergeRaster retains the values of the tile region, over the values in any buffered regions. This is useful for reducing edge effects when performing raster operations involving contagious processes.

This function is parallel-aware using the same mechanism as used in raster: NOTE: This may not work as expected as we transition away from raster. Specifically, if you start a cluster using raster::beginCluster(), then this function will automatically use that cluster. It is always a good idea to stop the cluster when finished, using raster::endCluster().

Value

mergeRaster returns a RasterLayer object.

splitRaster returns a list (length nx*ny) of cropped raster tiles.

Author(s)

Yong Luo, Alex Chubaty, Tati Micheletti & Ian Eddy

Alex Chubaty and Yong Luo

See Also

terra::merge(), terra::mosaic()

do.call(), terra::merge().

Examples

```
library(terra)

origDTThreads <- data.table::setDTthreads(2L)
origNcpus <- options(Ncpus = 2L)
set.seed(1462)

## an example with dimensions: nrow = 77,  ncol = 101, nlayers = 3
b <- rast(system.file("ex/logo.tif", package = "terra"))
r <- b[[1]] # use first layer only
nx <- 3
ny <- 4

tmpdir <- dir.create(file.path(tempdir(), "splitRaster-example"), showWarnings = FALSE)

y0 <- splitRaster(r, nx, ny, path = file.path(tmpdir, "y0")) # no buffer

## buffer: 10 pixels along both axes
y1 <- splitRaster(r, nx, ny, c(10, 10), path = file.path(tmpdir, "y1"))

## buffer: half the width and length of each tile
y2 <- splitRaster(r, nx, ny, c(0.5, 0.5), path = file.path(tmpdir, "y2"))

## the original raster:
if (interactive()) plot(r) # may require a call to `dev()` if using RStudio

## the split raster:
layout(mat = matrix(seq_len(nx * ny), ncol = nx, nrow = ny))
plotOrder <- unlist(lapply(split(1:12, rep(1:nx, each = ny)), rev))

if (interactive()) {
  invisible(lapply(y0[plotOrder], terra::plot))
}

## parallel splitting
if (requireNamespace("raster", quietly = TRUE) &&
    requireNamespace("parallel")) {
  if (interactive()) {
    n <- pmin(parallel::detectCores(), 4) # use up to 4 cores
    raster::beginCluster(n, type = "PSOCK")
    y3 <- splitRaster(r, nx, ny, c(0.7, 0.7), path = file.path(tmpdir, "y3"))
    raster::endCluster()
    if (interactive()) {
      invisible(lapply(y3[plotOrder], terra::plot))
    }
  }
}

## can be recombined using `terra::merge`
m0 <- do.call(merge, y0)
all.equal(m0, r) ## TRUE
```

```
m1 <- do.call(merge, y1)
all.equal(m1, r) ## TRUE

m2 <- do.call(merge, y2)
all.equal(m2, r) ## TRUE

## or recombine using mergeRaster
n0 <- mergeRaster(y0)
all.equal(n0, r) ## TRUE

n1 <- mergeRaster(y1)
all.equal(n1, r) ## TRUE

n2 <- mergeRaster(y2)
all.equal(n2, r) ## TRUE

# clean up
data.table::setDTthreads(origDTThreads)
options(Ncpus = origNcpus)
unlink(tmpdir, recursive = TRUE)
```

`middlePixel` *Return the (approximate) middle pixel on a raster*

Description

Return the (approximate) middle pixel on a raster

Usage

```
middlePixel(ras)
```

Arguments

`ras`	A `Raster` or `SpatRaster` object

Value

This calculation is slightly different depending on whether the `nrow(ras)` and `ncol(ras)` are even or odd. It will return the exact middle pixel if these are odd, and the pixel just left and/or above the middle pixel if either dimension is even, respectively.

move *Move*

Description

Wrapper for selecting different animal movement methods.

This version uses just turn angles and step lengths to define the correlated random walk.

Usage

```
move(hypothesis = "crw", ...)

crw(
  agent,
  extent,
  stepLength,
  stddev,
  lonlat = FALSE,
  torus = FALSE,
  returnMatrix = FALSE
)
```

Arguments

hypothesis	Character vector, length one, indicating which movement hypothesis/method to test/use. Currently defaults to 'crw' (correlated random walk) using crw.
...	arguments passed to the function in hypothesis
agent	A SpatVector points geometry, a two-column matrix of coordinates, or a SpatialPoints* object (deprecated; it is converted to a SpatVector on entry and converted back on return, so it needs sf and sp). If is has attributes, e.g., SpatialPointsDataFrame, 2 of the columns must be x1 and y1, indicating the previous location. If it does not have these columns as attributes, x1 and y1 will be assigned randomly.
extent	An optional Extent object that will be used for torus.
stepLength	Numeric vector of length 1 or number of agents describing step length.
stddev	Numeric vector of length 1 or number of agents describing standard deviation of wrapped normal turn angles.
lonlat	Logical. If TRUE, coordinates should be in degrees. If FALSE coordinates represent planar ('Euclidean') space (e.g. units of meters)
torus	Logical. Should the movement be wrapped to the opposite side of the map, as determined by the extent argument. Default FALSE.
returnMatrix	If TRUE then the return object will be a matrix. This will be MUCH faster than retaining the sp or SpatVector class, and thus will be much more effective for iterative crw calls

Details

This simple version of a correlated random walk is largely the version that was presented in Turchin 1998, but it was also used with bias modifications in McIntire, Schultz, Crone 2007.

Value

A SpatVector points object with updated spatial position defined by a single occurrence of step length(s) and turn angle(s).

Author(s)

Eliot McIntire

References

Turchin, P. 1998. Quantitative analysis of movement: measuring and modeling population redistribution in animals and plants. Sinauer Associates, Sunderland, MA.

McIntire, E. J. B., C. B. Schultz, and E. E. Crone. 2007. Designing a network for butterfly habitat restoration: where individuals, populations and landscapes interact. Journal of Applied Ecology 44:725-736.

See Also

terra::distance()

wrapTorus()

Examples

```
origDTThreads <- data.table::setDTthreads(2L)
origNcpus <- options(Ncpus = 2L)

# using just matrix
N <- 10
xrange <- yrange <- c(-50, 50)
starts <- cbind(x = stats::runif(N, xrange[1], xrange[2]),
                y = stats::runif(N, yrange[1], yrange[2]))
moved <- crw(starts, stepLength = 5, stddev = 10)
plot(starts, col = rainbow(10), pch = 19)
points(moved, col = rainbow(10))

# as SpatVector
agent <- terra::vect(starts)
moved <- crw(agent, stepLength = 5, stddev = 10)
movedAgain <- crw(moved, stepLength = 5, stddev = 10)
terra::plot(agent)
terra::plot(moved, add = TRUE, col = "red")
terra::plot(movedAgain, add = TRUE, col = "green")

# 1000x faster!! -- returnMatrix = TRUE
agentOrig <- agent
```

```
reps <- 1e2
system.time({
  for (i in 1:reps) agent <- crw(agent, stepLength = 5, stddev = 10, returnMatrix = TRUE)
})
agent <- agentOrig
system.time({
  for (i in 1:reps) agent <- crw(agent, stepLength = 5, stddev = 10)
})

# as sp (deprecated: converted to a SpatVector internally and back on return)
if (requireNamespace("sp") && requireNamespace("sf")) {
  agent <- sp::SpatialPoints(starts)
  spdf <- crw(agent, stepLength = 5, stddev = 10)
  spdfNew <- crw(spdf, stepLength = 5, stddev = 10)
  terra::plot(spdf, pch = 19)
  terra::points(spdfNew, col = "blue", pch = 19)
}

# clean up
data.table::setDTthreads(origDTThreads)
options(Ncpus = origNcpus)
```

`neutralLandscapeMap` *Produce a neutral (random) landscape raster*

Description

Generates a `SpatRaster` of randomly-varying values on the same grid as a template raster `x`, using a built-in Gaussian-smoothed random field with no extra package dependencies.

Usage

```
neutralLandscapeMap(
  x,
  pad = 10L,
  type = "gaussian",
  smooth = 3L,
  rescale = TRUE,
  ...
)
```

Arguments

`x`	A `SpatRaster` (or `RasterLayer`) to use as a template for the grid and CRS.
`pad`	Integer. Number of cells by which to pad `x` internally; the padding is cropped from the result.
`type`	Character. Only `"gaussian"` (the default) is supported.

smooth	Half-width (in cells) of the smoothing kernel; larger values produce smoother landscapes with longer-range autocorrelation.
rescale	If TRUE (default), rescale output values to [0, 1].
...	Currently ignored; accepted for backwards compatibility.

Details

The "gaussian" generator fills a padded grid with i.i.d. normal noise, then smooths it with a square mean kernel of size 2 * smooth + 1. The result is a roughly Gaussian random field, suitable for sample modules, demos, and tests.

Value

A SpatRaster (or RasterLayer, matching x) with the same extent, resolution, and CRS as x, filled with randomly-generated values.

Examples

```
library(terra)
nx <- ny <- 100L
r <- rast(nrows = ny, ncols = nx,
          xmin = -nx/2, xmax = nx/2,
          ymin = -ny/2, ymax = ny/2)

map_default <- neutralLandscapeMap(r)
if (interactive()) plot(map_default)

## Tweak smoothness:
map_rough  <- neutralLandscapeMap(r, smooth = 1)
map_smooth <- neutralLandscapeMap(r, smooth = 8)
```

---

numAgents *SELES - Number of Agents to initiate*

Description

Sets the the number of agents to initiate. THIS IS NOT YET FULLY IMPLEMENTED.

A SELES-like function to maintain conceptual backwards compatibility with that simulation tool. This is intended to ease transitions from SELES.

You must know how to use SELES for these to be useful.

Usage

```
numAgents(N, probInit)
```

Arguments

N	Number of agents to initiate (integer scalar).
probInit	Probability of initializing an agent at the location.

Value

A numeric, indicating number of agents to start

Author(s)

Eliot McIntire

---

patchSize *Patch size*

Description

Patch size

Usage

```
patchSize(patches)
```

Arguments

patches	Number of patches.

---

probInit SELES - *Probability of Initiation*

Description

Describes the probability of initiation of agents or events. *THIS IS NOT FULLY IMPLEMENTED.*

A SELES-like function to maintain conceptual backwards compatibility with that simulation tool. This is intended to ease transitions from SELES.

You must know how to use SELES for these to be useful.

Usage

```
probInit(map, p = NULL, absolute = NULL)
```

Arguments

map	A spatialObjects object. Currently, only provides CRS and, if p is not a raster, then all the raster dimensions.
p	probability, provided as a numeric or raster
absolute	logical. Is p absolute probabilities or relative?

Value

A RasterLayer with probabilities of initialization. There are several combinations of inputs possible and they each result in different behaviours.

If p is numeric or Raster and between 0 and 1, it is treated as an absolute probability, and a map will be produced with the p value(s) everywhere.

If p is numeric or Raster and not between 0 and 1, it is treated as a relative probability, and a map will be produced with p/max(p) value(s) everywhere.

If absolute is provided, it will override the previous statements, unless absolute = TRUE and p is not between 0 and 1 (i.e., is not a probability).

Author(s)

Eliot McIntire

---

randomPolygons *Produce a* SpatRaster *of random polygons*

Description

These are built with the spread() function internally.

Produces a SpatVector polygons object with 1 feature that will have approximately an area equal to area (expecting area in hectares), #' and a centre at approximately x.

Usage

```
randomPolygons(
  ras = rast(ext(0, 15, 0, 15), resolution = 1, vals = 0),
  numTypes = 2,
  ...
)

randomPolygon(x, hectares, area)

## Default S3 method:
randomPolygon(x, hectares, area)
```

Arguments

ras	A raster that whose extent will be used for the random polygons.
numTypes	Numeric value. The number of unique polygon types to use.
...	Other arguments passed to spread. No known uses currently.
x	Either a SpatVector, or SpatialPoints (deprecated), SpatialPolygons (deprecated), or matrix with two dimensions, 1 row, with the approximate centre of the new random polygon to create. If matrix, then longitude and latitude are assumed (epsg:4326).
hectares	Deprecated. Use area in meters squared.
area	A numeric, the approximate area in meters squared of the random polygon.

Value

A map of extent ext with random polygons.

A SpatVector polygons object, with approximately the area request, centred approximately at the coordinates requested, in the projection of x.

See Also

spread(), randomPolygons()

Examples

```
origDTThreads <- data.table::setDTthreads(2L)
origNcpus <- options(Ncpus = 2L)

set.seed(1234)
Ras <- randomPolygons(numTypes = 5)
if (interactive() ) {
  terra::plot(Ras, col = c("yellow", "dark green", "blue", "dark red"))
}

# more complex patterning, with a range of patch sizes
r <- terra::rast(terra::ext(0, 50, 0, 50), resolution = 1, vals = 0)
a <- randomPolygons(numTypes = 400, r)
a[a < 320] <- 0
a[a >= 320] <- 1
clumped <- terra::patches(a)
if (interactive()) {
  terra::plot(a)
}

# clean up
data.table::setDTthreads(origDTThreads)
options(Ncpus = origNcpus)

latLong <- terra::crs("+proj=longlat +datum=WGS84 +no_defs")
# "epsg:4326"
a1 <- terra::vect(cbind(-110, 59), crs = latLong)
```

```
a2 <- randomPolygon(a1, area = 1e7)

if (interactive()) {
  terra::plot(a1)
  terra::points(a2, pch = 19)
}

if (require("sf", quietly = TRUE)) {
  latLong <- sf::st_crs("+proj=longlat +datum=WGS84 +no_defs")
  # "epsg:4326"

  b1 <- list(cbind(
    x = c(-122.98, -116.1, -99.2, -106, -122.98),
    y = c(59.9, 65.73, 63.58, 54.79, 59.9)
  )) |>
    sf::st_polygon() |>
    sf::st_sfc() |>
    sf::st_sf(geometry = _, ID = 1L, shinyLabel = "zone2", crs = latLong)
  b2 <- randomPolygon(b1, area = 1e10)

  if (interactive()) {
    plot(sf::st_geometry(b1))
    plot(sf::st_geometry(b2), add = TRUE)
  }
}
```

`randomStudyArea` *Create default study areas for use with* `SpaDES` *modules*

Description

Create default study areas for use with `SpaDES` modules

Usage

```
randomStudyArea(center = NULL, size = 10000, seed = NULL)
```

Arguments

`center`	`SpatialPoints` object specifying a set of coordinates and a projection. Default is an area in southern Alberta, Canada.
`size`	Numeric specifying the approximate size of the area in m^2. Default `1e4`.
`seed`	Numeric indicating the random seed to set internally (useful for ensuring the same study area is produced each time).

Value

`SpatVector`

Examples

```
a <- randomStudyArea(seed = 123)
if (interactive()) {
  terra::plot(a)
}
```

`rasterizeReduced` *Convert reduced representation to full raster*

Description

Convert reduced representation to full raster

Usage

```
rasterizeReduced(
  reduced,
  fullRaster,
  newRasterCols,
  mapcode = names(fullRaster),
  ...
)
```

Arguments

`reduced`	`data.frame` or `data.table` that has at least one column of codes that are represented in the `fullRaster`.
`fullRaster`	`RasterLayer/SpatRaster` of codes used in `reduced` that represents a spatial representation of the data. Note that if `fullRaster` is a `factor SpatRaster`, the active category level values are used, not the IDs (see `terra::activeCat` and `terra::cats`)
`newRasterCols`	Character vector, length 1 or more, with the name(s) of the column(s) in `reduced` whose value will be returned as a `RasterLayer/SpatRaster` or list of `RasterLayer/SpatRasters`.
`mapcode`	a character, length 1, with the name of the column in `reduced` that is represented in `fullRaster`.
`...`	Other arguments. None used yet.

Value

A `RasterLayer/SpatRaster` or list of `RasterLayer/SpatRaster` of with same dimensions as `fullRaster` representing `newRasterCols` spatially, according to the join between the `mapcode` contained within `reduced` and `fullRaster`

Author(s)

Eliot McIntire

See Also

terra::rast()

Examples

```
library(data.table)
library(terra)

origDTThreads <- data.table::setDTthreads(2L)
origNcpus <- options(Ncpus = 2L)

ras <- rast(ext(0, 15, 0, 15), res = 1)
fullRas <- randomPolygons(ras, numTypes = 2)
names(fullRas) <- "mapcodeAll"
uniqueComms <- unique(fullRas)
reducedDT <- data.table(uniqueComms,
                        communities = sample(1:1000, length(uniqueComms)),
                        biomass = rnbinom(length(uniqueComms), mu = 4000, 0.4))
biomass <- rasterizeReduced(reducedDT, fullRas, "biomass")

# The default key is the layer name of the fullRas, so rekey incase of miskey
setkey(reducedDT, biomass)

communities <- rasterizeReduced(reducedDT, fullRas, "communities")
coltab(communities) <- c("blue", "orange", "red")
if (interactive()) {
  terra::plot(c(biomass, communities, fullRas))
}

## with a factor SpatRaster, the mapcode should correspond to the
## active category (not the ids)
cls <- data.frame(id = sort(unique(as.vector(fullRas[]))))
cls$Bclass <- LETTERS[cls$id]
levels(fullRas) <- cls
is.factor(fullRas)

clsDT <- as.data.table(cls)
reducedDT <- reducedDT[clsDT, on = "mapcodeAll==id"]
reducedDT[, mapcodeAll := Bclass]

biomass2 <- rasterizeReduced(reducedDT, fullRas, "biomass")

# clean up
data.table::setDTthreads(origDTThreads)
options(Ncpus = origNcpus)
```

---

rastFromDF *Build a multi-layer SpatRaster from a sparse cell/value table*

Description

Efficiently constructs a multi-layer terra::SpatRaster from a `data.frame`/`data.table` that contains one column of raster cell identifiers and one or more columns of values. The function performs a complete join against all cell indices in the template raster (so every cell is present exactly once), fills missing cells with `NA`, and assigns all layers in a single block write for speed.

Usage

```
rastFromDF(df, rasTemplate, cols = NULL, cellIDcol = NULL)
```

Arguments

`df`	A `data.frame` or `data.table` containing one column for raster cell indices (name matching `"pixel|cell"`; e.g., `cell`, `cellID`, `pixelID`) and one or more columns of values to become raster layers.
`rasTemplate`	A terra::SpatRaster whose geometry (extent, resolution, rows/cols, CRS) defines the output grid.
`cols`	`NULL` or a character vector of column names in `df` to use as layers (order preserved). If `NULL`, the function uses all columns except the detected cell/pixel ID column.
`cellIDcol`	A single string with the column name of the cellID/pixelID. Default is `NULL`, which will use the first column that matches a grep on `"pixel|cell"`.

Details

This function is designed as a fast "inverse" of `terra::as.data.frame()` for rasters to be made from "sparse" data.frame/data.table objects with the following workflow:

1. Identify the pixel/cell ID column in `df` by searching for a column name that matches `"pixel|cell"` (e.g., `"cell"`, `"cellID"`, `"pixelID"`).
2. Create a complete index of all cell numbers for `rasTemplate` using `terra::ncell()` and join with `df`, so any non-provided cells become `NA`.
3. Create a multi-layer raster with the same geometry as `rasTemplate` via `terra::rast()`, one layer per value column, and write all values at once using `terra::values()`.

Notes and gotchas

- The ID column must be a 1-based integer cell index consistent with `rasTemplate`'s grid.
- If the input has duplicate cell IDs, the data.table join semantics will replicate rows; the final `terra::values()` call expects exactly one row per cell. Deduplicate beforehand if necessary.
- The argument `cols` (a character vector) can be used to select a subset of columns to create in the returned SpatVector.
- The number of layers created equals the number of selected value columns. Layer names follow the column names.

Value

A terra::SpatRaster with one layer per selected value column. Cells not present in `df` are filled with `NA`. The raster shares the geometry of `rasTemplate`.

Examples

```
library(terra)
library(data.table)

# Template raster (3x3 grid)
tmpl <- rast(nrows = 3, ncols = 3, xmin = 0, xmax = 3, ymin = 0, ymax = 3,
             crs = "EPSG:4326")

# Sparse table: provide values for a subset of cells
df <- data.table(
  cell = c(1L, 3L, 5L, 9L),
  v1   = c(10, 20, 30, 40),
  v2   = c(100, 200, 300, 400)
)

r <- rastFromDF(df, tmpl)
r
terra::plot(r)  # remaining cells are NA
```

`rings` *Identifies all cells within a ring around the focal cells*

Description

Identifies the cell numbers of all cells within a ring defined by minimum and maximum distances from focal cells. Uses `spread()` under the hood, with specific values set. Under many situations, this may be faster than using `sf::st_buffer` twice (once for smaller ring and once for larger ring, then removing the smaller ring cells).

Usage

```
rings(
  landscape,
  loci = NA_real_,
  id = FALSE,
  minRadius = 2,
  maxRadius = 5,
  allowOverlap = FALSE,
  returnIndices = FALSE,
  returnDistances = TRUE,
  ...
)
```

Arguments

`landscape`	A `RasterLayer` or `SpatRaster` object. This defines the possible locations for spreading events to start and spread into. This can also be used as part of `stopRule`.

loci	A vector of locations in landscape. These should be cell indices. If user has x and y coordinates, these can be converted with cellFromXY().
id	Logical. If TRUE, returns a raster of events ids. If FALSE, returns a raster of iteration numbers, i.e., the spread history of one or more events. NOTE: this is overridden if returnIndices is TRUE or 1 or 2.
minRadius	Numeric. Minimum radius to be included in the ring. Note: this is inclusive, i.e., >=.
maxRadius	Numeric. Maximum radius to be included in the ring. Note: this is inclusive, i.e., <=.
allowOverlap	Logical. If TRUE, then individual events can overlap with one another, i.e., they do not interact (this is slower than if allowOverlap = FALSE). Default is FALSE.
returnIndices	Logical or numeric. If 1 or TRUE, will return a data.table with indices and values of successful spread events. If 2, it will simply return a vector of pixel indices of all cells that were touched. This will be the fastest option. If FALSE, then it will return a raster with values. See Details.
returnDistances	Logical. Should the function include a column with the individual cell distances from the locus where that event started. Default is FALSE. See Details.
...	Any other argument passed to spread

Value

This will return a data.table with columns as described in spread when returnIndices = TRUE.

Author(s)

Eliot McIntire

See Also

cir() which uses a different algorithm. cir tends to be faster when there are few starting points, rings tends to be faster when there are many starting points. Another difference between the two functions is that rings takes the centre of the pixel as the centre of a circle, whereas cir takes the exact coordinates. See example.

sf::st_buffer

Examples

```
library(terra)

origDTThreads <- data.table::setDTthreads(2L)
origNcpus <- options(Ncpus = 2L)
set.seed(1462)

# Make random forest cover map
emptyRas <- terra::rast(terra::ext(0, 1e2, 0, 1e2), res = 1)

# start from two cells near middle
```

```
loci <- (ncell(emptyRas) / 2 - ncol(emptyRas)) / 2 + c(-3, 3)

# No overlap is default, occurs randomly
emptyRas[] <- 0
rngs <- rings(emptyRas, loci = loci, minRadius = 7, maxRadius = 9, returnIndices = TRUE)
emptyRas[rngs$indices] <- rngs$id
if (interactive()) {
  terra::plot(emptyRas)
}

# Variable ring widths, including centre cell for smaller one
emptyRas[] <- 0
rngs <- rings(emptyRas, loci = loci, minRadius = c(0, 7), maxRadius = c(8, 18),
              returnIndices = TRUE)
emptyRas[rngs$indices] <- rngs$id
if (interactive()) {
  terra::plot(emptyRas)
}

# clean up
data.table::setDTthreads(origDTThreads)
options(Ncpus = origNcpus)
```

---

runifC *Rcpp Sugar version of runif*

Description

Slightly faster than runif, and used a lot

Usage

```
runifC(N)
```

Arguments

N	Integer Vector

Value

A vector of uniform random numbers as per runif

specificNumPerPatch *Initiate a specific number of agents in a map of patches*

Description

Instantiate a specific number of agents per patch. The user can either supply a table of how many to initiate in each patch, linked by a column in that table called pops.

Usage

```
specificNumPerPatch(patches, numPerPatchTable = NULL, numPerPatchMap = NULL)
```

Arguments

patches	SpatRaster of patches, with some sort of a patch id.
numPerPatchTable	A data.frame or data.table with a column named pops that matches the patches patch ids, and a second column num.in.pop with population size in each patch.
numPerPatchMap	A SpatRaster exactly the same as patches but with agent numbers rather than ids as the cell values per patch.

Value

A raster with 0s and 1s, where the 1s indicate starting locations of agents following the numbers above.

Examples

```
library(data.table)

origDTThreads <- data.table::setDTthreads(2L)
origNcpus <- options(Ncpus = 2L)

set.seed(1234)
Ntypes <- 4
ras <- randomPolygons(numTypes = Ntypes)
if (interactive()) {
  terra::plot(ras)
}

# Use numPerPatchTable
patchDT <- data.table(pops = 1:Ntypes, num.in.pop = c(1, 3, 5, 7))
rasAgents <- specificNumPerPatch(ras, patchDT)
rasAgents[is.na(rasAgents)] <- 0

if (require(testthat))
  expect_true(all(unname(table(ras[rasAgents])) == patchDT$num.in.pop))
```

```
# Use numPerPatchMap
# Remap in one pass: rewriting in place would collide, because a patch
# rewritten to a value that is also a later patch's id gets overwritten
# again on that iteration.
rasPatches <- ras
terra::values(rasPatches) <-
  patchDT$num.in.pop[match(terra::values(ras)[, 1], patchDT$pops)]
if (interactive()) {
  terra::plot(c(ras, rasPatches))
}
rasAgents <- specificNumPerPatch(ras, numPerPatchMap = rasPatches)
rasAgents[is.na(rasAgents)] <- 0
if (interactive()) {
  terra::plot(rasAgents)
}

# clean up
data.table::setDTthreads(origDTThreads)
options(Ncpus = origNcpus)
```

---

spokes *Identify outward radiating spokes from initial points*

Description

This is a generalized version of a notion of a viewshed. The main difference is that there can be many "viewpoints".

Usage

```
spokes(
  landscape,
  coords,
  loci,
  maxRadius = ncol(landscape)/4,
  minRadius = maxRadius,
  allowOverlap = TRUE,
  stopRule = NULL,
  includeBehavior = "includePixels",
  returnDistances = FALSE,
  angles = NA_real_,
  nAngles = NA_real_,
  returnAngles = FALSE,
  returnIndices = TRUE,
  ...
)
```

Arguments

landscape	Raster on which the circles are built.
coords	Either a matrix with 2 (or 3) columns, x and y (and id), representing the coordinates (and an associated id, like cell index), or a SpatialPoints* object around which to make circles. Must be same coordinate system as the landscape argument. Default is missing, meaning it uses the default to loci.
loci	Numeric. An alternative to coords. These are the indices on landscape to initiate this function (see coords). Default is one point in centre of landscape.
maxRadius	Numeric vector of length 1 or same length as coords
minRadius	Numeric vector of length 1 or same length as coords. Default is maxRadius, meaning return all cells that are touched by the narrow ring at that exact radius. If smaller than maxRadius, then this will create a buffer or doughnut or ring.
allowOverlap	Logical. Should duplicates across id be removed or kept. Default TRUE.
stopRule	A function. If the spokes are to stop. This can be a function of landscape, fromCell, toCell, x (distance from coords cell), or any other named argument passed into the ... of this function. See examples.
includeBehavior	Character string. Currently accepts only "includePixels", the default, and "excludePixels". See details.
returnDistances	Logical. If TRUE, then a column will be added to the returned data.table that reports the distance from coords to every point that was in the circle/doughnut surrounding coords. Default FALSE, which is faster.
angles	Numeric. Optional vector of angles, in radians, to use. This will create "spokes" outward from coords. Default is NA, meaning, use internally derived angles that will "fill" the circle.
nAngles	Numeric, length one. Alternative to angles. If provided, the function will create a sequence of angles from 0 to 2*pi, with a length nAngles, and not including 2*pi. Will not be used if angles is provided, and will show warning of both are given.
returnAngles	Logical. If TRUE, then a column will be added to the returned data.table that reports the angle from coords to every point that was in the circle/doughnut surrounding coords. Default FALSE.
returnIndices	Logical or numeric. If 1 or TRUE, will return a data.table with indices and values of successful spread events. If 2, it will simply return a vector of pixel indices of all cells that were touched. This will be the fastest option. If FALSE, then it will return a raster with values. See Details.
...	Objects to be used by stopRule(). See examples.

Value

A matrix containing columns id (representing the row numbers of coords), angles (from coords to each point along the spokes), x and y coordinates of each point along the spokes, the corresponding indices on the landscape Raster, dists (the distances between each coords and each point along the spokes), and stop, indicating if it was a point that caused a spoke to stop going outwards due to stopRule.

Author(s)

Eliot McIntire

Examples

```
library(data.table)
library(terra)

origDTThreads <- data.table::setDTthreads(2L)
origNcpus <- options(Ncpus = 2L)
set.seed(1234)

ras <- terra::rast(terra::ext(0, 10, 0, 10), res = 1, val = 0)
rp <- randomPolygons(ras, numTypes = 10)

if (interactive())
  terra::plot(rp)

angles <- seq(0, pi * 2, length.out = 17)
angles <- angles[-length(angles)]
n <- 2
loci <- sample(terra::ncell(rp), n)
coords <- terra::vect(terra::xyFromCell(rp, loci))
stopRule <- function(landscape) landscape < 3
d2 <- spokes(rp, coords = coords, stopRule = stopRule,
             minRadius = 0, maxRadius = 50,
             returnAngles = TRUE, returnDistances = TRUE,
             allowOverlap = TRUE, angles = angles, returnIndices = TRUE)

# Assign values to the "patches" that were in the viewshed of a ray
rasB <- terra::rast(ras)
rasB[] <- 0
rasB[d2[d2[, "stop"] == 1, "indices"]] <- 1

if (interactive()) {
  rasB[rasB == 0] <- NA
  terra::plot(rasB, add = TRUE, col = "red", legend = FALSE)
}

if (NROW(d2) > 0) {
  sp1 <- terra::vect(d2[, c("x", "y")])
  if (interactive())
    terra::plot(sp1, add = TRUE, pch = 19)
}
if (interactive())
  terra::plot(coords, add = TRUE, pch = 19, col = "blue")

# clean up
data.table::setDTthreads(origDTThreads)
options(Ncpus = origNcpus)
```

spread *Simulate a spread process on a landscape.*

Description

This can be used to simulate fires, seed dispersal, calculation of iterative, concentric landscape values (symmetric or asymmetric) and many other things. Essentially, it starts from a collection of cells (`loci`) and spreads to neighbours, according to the `directions` and `spreadProb` arguments. This can become quite general, if `spreadProb` is 1 as it will expand from every loci until all cells in the landscape have been covered. With `id` set to `TRUE`, the resulting map will be classified by the index of the cell where that event propagated from. This can be used to examine things like fire size distributions. **NOTE:** See also `spread2()`, which is more robust and can be used to build custom functions. However, under some conditions, this `spread` function is faster. The two functions can accomplish many of the same things, but their argument names and, in particular, their **output column names differ**; see the section "Differences between `spread` and `spread2`" below before converting code from one to the other.

Usage

```
spread(
  landscape,
  loci = NA_real_,
  spreadProb = 0.23,
  persistence = 0,
  mask = NA,
  maxSize = 100000000L,
  directions = 8L,
  iterations = 1000000L,
  lowMemory = NULL,
  returnIndices = FALSE,
  returnDistances = FALSE,
  mapID = NULL,
  id = FALSE,
  plot.it = FALSE,
  spreadProbLater = NA_real_,
  spreadState = NA,
  circle = FALSE,
  circleMaxRadius = NA_real_,
  stopRule = NA,
  stopRuleBehavior = "includeRing",
  allowOverlap = FALSE,
  asymmetry = NA_real_,
  asymmetryAngle = NA_real_,
  quick = FALSE,
  skipChecks = quick,
  neighProbs = NULL,
  exactSizes = FALSE,
```

```
  relativeSpreadProb = FALSE,
  ...
)
```

Arguments

landscape	A RasterLayer or SpatRaster object. This defines the possible locations for spreading events to start and spread into. This can also be used as part of stopRule.
loci	A vector of locations in landscape. These should be cell indices. If user has x and y coordinates, these can be converted with cellFromXY().
spreadProb	Numeric, RasterLayer, or SpatRaster. If numeric of length 1, then this is the global probability of spreading into each cell from a neighbour. If a raster (or a vector of length terra::ncell(landscape), resolution and extent of landscape), then this will be the cell-specific probability. Default is 0.23. If a spreadProbLater is provided, then this is only used for the first iteration. Also called "escape probability". See section on "Breaking out of spread events".
persistence	A length 1 probability that an active cell will continue to burn, per time step.
mask	RasterLayer or SpatRaster object congruent with landscape, whose elements are 0, 1, where 1 indicates "cannot spread to". Currently not implemented, but identical behaviour can be achieved if spreadProb has zeros in all unspreadable locations.
maxSize	Numeric. Maximum number of cells for a single or all events to be spread. Recycled to match loci length, if it is not as long as loci. See section on Breaking out of spread events.
directions	The number of adjacent cells in which to look; default is 8 (Queen case). Can only be 4 or 8.
iterations	Number of iterations to spread. Leaving this NULL allows the spread to continue until stops spreading itself (i.e., exhausts itself).
lowMemory	Deprecated.
returnIndices	Logical or numeric. If 1 or TRUE, will return a data.table with indices and values of successful spread events. If 2, it will simply return a vector of pixel indices of all cells that were touched. This will be the fastest option. If FALSE, then it will return a raster with values. See Details.
returnDistances	Logical. Should the function include a column with the individual cell distances from the locus where that event started. Default is FALSE. See Details.
mapID	Deprecated. Use id.
id	Logical. If TRUE, returns a raster of events ids. If FALSE, returns a raster of iteration numbers, i.e., the spread history of one or more events. NOTE: this is overridden if returnIndices is TRUE or 1 or 2.
plot.it	If TRUE, then plot the raster at every iteration, so one can watch the spread event grow.
spreadProbLater	Numeric, or RasterLayer. If provided, then this will become the spreadProb after the first iteration. See Details.

spreadState — data.table. This should be the output of a previous call to spread, where returnIndices was TRUE. Default NA, meaning the spread is starting from loci. See Details.

circle — Logical. If TRUE, then outward spread will be by equidistant rings, rather than solely by adjacent cells (via directions arg.). Default is FALSE. Using circle = TRUE can be dramatically slower for large problems. Note, this should usually be used with spreadProb = 1.

circleMaxRadius — Numeric. A further way to stop the outward spread of events. If circle is TRUE, then it will grow to this maximum radius. See section on Breaking out of spread events. Default is NA.

stopRule — A function which will be used to assess whether each individual cluster should stop growing. This function can be an argument of "landscape", "id", "cells", and any other variables passed to spread in the .... cells and landscape will both be numeric vectors of length of active cells. cells will be the raster index, so can be used to extract values from another raster passed via ... . Default NA, meaning that spreading will not stop as a function of the landscape. See section on "Breaking out of spread events" and examples.

stopRuleBehavior — Character. Can be one of "includePixel", "excludePixel", "includeRing", or "excludeRing". If stopRule contains a function, this argument is used determine what to do with the cell(s) that caused the rule to be TRUE. See details. Default is "includeRing" which means to accept the entire ring of cells that caused the rule to be TRUE.

allowOverlap — Logical. If TRUE, then individual events can overlap with one another, i.e., they do not interact (this is slower than if allowOverlap = FALSE). Default is FALSE.

asymmetry — A numeric indicating the ratio of the asymmetry to be used. Default is NA, indicating no asymmetry. See details. This is still experimental. **Use with caution.**

asymmetryAngle — A numeric indicating the angle in degrees (0 is "up", as in North on a map), that describes which way the asymmetry is.

quick — Logical. If TRUE, then several potentially time consuming checking (such as inRange) will be skipped. This should only be used if there is no concern about checking to ensure that inputs are legal.

skipChecks — An alias for quick, the name used by spread2() and spread3(). Supplying either has the same effect; supplying skipChecks used to be silently ignored here.

neighProbs — A numeric vector, whose sum is 1. It indicates the probabilities an individual spread iteration spreading to 1:length(neighProbs) neighbours.

exactSizes — Logical. If TRUE, then the maxSize will be treated as exact sizes, i.e., the spread events will continue until they are floor(maxSize). This is overridden by iterations, but if iterations is run, and individual events haven't reached maxSize, then the returned data.table will still have at least one active cell per event that did not achieve maxSize, so that the events can continue if passed into spread with spreadState.

relativeSpreadProb
: Logical. If TRUE, then spreadProb will be rescaled *within* the directions neighbours, such that the sum of the probabilities of all neighbours will be 1. Default FALSE, unless spreadProb values are not contained between 0 and 1, which will force relativeSpreadProb to be TRUE.

...
: Additional named vectors or named list of named vectors required for stopRule. These vectors should be as long as required e.g., length loci if there is one value per event.

Details

For large rasters, a combination of lowMemory = TRUE and returnIndices = TRUE or returnIndices = 2 will be fastest and use the least amount of memory. **2022-07-25:** lowMemory = TRUE **is deprecated due to removal of package** ffbase **from CRAN.**

This function can be interrupted before all active cells are exhausted if the iterations value is reached before there are no more active cells to spread into. If this is desired, returnIndices should be TRUE and the output of this call can be passed subsequently as an input to this same function. This is intended to be used for situations where external events happen during a spread event, or where one or more arguments to the spread function change before a spread event is completed. For example, if it is desired that the spreadProb change before a spread event is completed because, for example, a fire is spreading, and a new set of conditions arise due to a change in weather.

asymmetry is currently used to modify the spreadProb in the following way. First for each active cell, spreadProb is converted into a length 2 numeric of Low and High spread probabilities for that cell: spreadProbsLH <- (spreadProb*2) // (asymmetry+1)*c(1,asymmetry), whose ratio is equal to asymmetry. Then, using asymmetryAngle, the angle between the initial starting point of the event and all potential cells is found. These are converted into a proportion of the angle from -asymmetryAngle to asymmetryAngle using: angleQuality <- (cos(angles - rad2(asymmetryAngle))+1)/2 where rad2 <- function (degree) (degree * pi)/180

These are then converted to multiple spreadProbs by spreadProbs <- lowSpreadProb + (angleQuality * diff(spreadProbsLH)) To maintain an expected spreadProb that is the same as the asymmetric spreadProbs, these are then rescaled so that the mean of the asymmetric spreadProbs is always equal to spreadProb at every iteration: spreadProbs <- spreadProbs - diff(c(spreadProb, mean(spreadProbs)))

Value

Either a RasterLayer indicating the spread of the process in the landscape or a data.table if returnIndices is TRUE. If a RasterLayer, then it represents every cell in which a successful spread event occurred. For the case of, say, a fire this would represent every cell that burned. If allowOverlap is TRUE, This RasterLayer will represent the sum of the individual event ids (which are numerics seq_along(loci). This will generally be of minimal use because it won't be possible to distinguish if event 2 overlapped with event 5 or if it was just event 7.

If returnIndices is TRUE, then this function returns a data.table with columns:

id	an arbitrary ID 1:length(loci) identifying unique clusters of spread events, i.e., all cells that have been sp
initialLocus	the initial cell number of that particular spread event.
indices	The cell indices of cells that have been touched by the spread algorithm.
active	a logical indicating whether the cell is active (i.e., could still be a source for spreading) or not (no spreading

This will generally be more useful when `allowOverlap` is `TRUE`.

Breaking out of spread events

There are 4 ways for the spread to "stop" spreading. Here, each "event" is defined as all cells that are spawned from a single starting loci. So, one spread call can have multiple spreading "events". The ways outlines below are all acting at all times, i.e., they are not mutually exclusive. Therefore, it is the user's responsibility to make sure the different rules are interacting with each other correctly. Using `spreadProb` or `maxSize` are computationally fastest, sometimes dramatically so.

`spreadProb` Probabilistically, if spreadProb is low enough, active spreading events will stop. In practice, active sprea

`maxSize` This is the number of cells that are "successfully" turned on during a spreading event. This can be vector

`circleMaxRadius` If `circle` is TRUE, then this will be the maximum radius reached, and then the event will stop. This is v

`stopRule` This is a function that can use "landscape", "id", "cells", or any named vector passed into `spread` in the .

The spread function does not return the result of this `stopRule`. If, say, an event has both `circleMaxRadius` and `stopRule`, and it is the `circleMaxRadius` that caused the event spreading to stop, there will be no indicator returned from this function that indicates which rule caused the stop.

`stopRule` has many use cases. One common use case is evaluating a neighbourhood around a focal set of points. This provides, therefore, an alternative to the `terra::buffer()` function or `terra::focal()` function. In both of those cases, the window/buffer size must be an input to the function. Here, the resulting size can be emergent based on the incremental growing and calculating of the `landscape` values underlying the spreading event.

`stopRuleBehavior`

This determines how the `stopRule` should be implemented. Because spreading occurs outwards in concentric circles or shapes, one cell width at a time, there are 4 possible ways to interpret the logical inequality defined in `stopRule`. In order of number of cells included in resulting events, from most cells to fewest cells:

`"includeRing"` Will include the entire ring of cells that, as a group, caused `stopRule` to be `TRUE`.

`"includePixel"` Working backwards from the entire ring that caused the `stopRule` to be `TRUE`, this will iteratively random

`"excludePixel"` Like `"includePixel"`, but it will not add back the cell that causes `stopRule` to be `TRUE`

`"excludeRing"` Analogous to `"excludePixel"`, but for the entire final ring of cells added. This will exclude the entire rin

Using `spreadCpp()` instead

With `options(spades.useSpreadCpp = TRUE)`, a call to this function is handed to the faster `spreadCpp()`, with its arguments and return value converted, so existing code runs unchanged. Results differ from this function's for the same seed (see `spreadCpp()`). A call using an argument `spreadCpp()` does not support runs this function as usual, with a message once per session. `exactSizes = TRUE` becomes `minSize = maxSize`; a raster return is supported only with `id = TRUE`. `spreadCpp()` returns a finished spread, every cell inactive, so a call with `iterations` below its default, which stops partway to be resumed, runs this function. As in this function, an all-`NA` `maxSize` means no size limit, and `maxSize` is rounded down to whole cells.

Differences between `spread` and `spread2`

Which to use. `spread2()` is the more robust of the two: its behaviour is better tested, it can be called iteratively (passing its own output back in as `start`), and more combinations of its arguments work as documented. `spread()` is generally faster, sometimes substantially so, but several of its argument combinations are untested, deprecated, or unimplemented.

Argument correspondence. Arguments with the same name in both functions generally mean the same thing, with the exceptions noted below.

`spread`	`spread2`
`loci`	`start`
`persistence`	`persistProb`
`returnIndices`	`asRaster`
`exactSizes`	`exactSize`
`spreadState`	`start`
`quick`, `skipChecks`	`skipChecks`
`relativeSpreadProb`	`spreadProbRel`
`allowOverlap`	`allowOverlap`
`maxSize`	`maxSize`
`mask`, `mapID`, `lowMemory`	–
`id`, `stopRule`, `stopRuleBehavior`, `circleMaxRadius`	–
–	`returnDirections`, `returnFrom`, `oneNeighbourOnly`, `maxRetries`

Output column names. This is the most common source of errors when converting code. The returned `data.tables` share *no* column names:

Meaning	`spread(returnIndices = TRUE)`	`spread2(asRaster = FALSE)`
Cell touched by the spread	`indices`	`pixels`
Cell the event started from	`initialLocus`	`initialPixels`
Whether the cell can still spread	`active` (logical)	`state` (character: `"activeSource"`, `"success`
Event identifier	`id`	not in the main table; see `attr(out, "spread`
Distance from origin	`dists` (only with `circle`/`allowOverlap`)	`distance` (with `returnDistances`)
Source of the previous step	–	`from` (with `returnFrom`)

Author(s)

Eliot McIntire and Steve Cumming

See Also

`spread2()` for a different implementation of the same algorithm. It is more robust, meaning, there will be fewer unexplainable errors, and the behaviour has been better tested, so it is more likely to be exactly as described under all argument combinations. Also, `rings()` which uses `spread` but with specific argument values selected for a specific purpose. `terra::distance()`. `cir()` to create "circles"; it is fast for many small problems.

Examples

```
library(terra)
origDTThreads <- data.table::setDTthreads(2L)
origNcpus <- options(Ncpus = 2L)

# Make random forest cover map
set.seed(123)
emptyRas <- rast(ext(0, 1e2, 0, 1e2), res = 1)
hab <- randomPolygons(emptyRas, numTypes = 40)
names(hab) <- "hab"
mask <- rast(emptyRas)
values(mask) <- 0
mask[1:5000] <- 1
numCol <- ncol(emptyRas)
numCell <- ncell(emptyRas)
directions <- 8

# Can use transparent as a colour
coltab(hab) <- paste(c("transparent", grey(0:40/40)))

terra::plot(hab)

# initiate 10 fires
startCells <- as.integer(sample(1:ncell(emptyRas), 100))
fires <- spread(hab, loci = startCells, 0.235, persistence = 0, numNeighs = 2,
                mask = NULL, maxSize = 1e8, directions = 8, iterations = 1e6, id = TRUE)

terra::plot(hab, type = "classes", legend = FALSE)
fires[fires == 0] <- NA
terra::plot(fires, add = TRUE, col = "red", type = "continuous", legend = FALSE)

# Instead, to give a colour to the zero values, use `zero.color=`
coltab(fires) <- NULL
# need to specify "type" to get correct legend
terra::plot(fires,  col = c(colorRampPalette(c("blue", "green"))(100)),
            type = "continuous")

##----------------------------------------------------------------------------
## Continue event by passing interrupted object into spreadState
##----------------------------------------------------------------------------

## Interrupt a spread event using iterations - need `returnIndices = TRUE` to
##  use outputs as new inputs in next iteration
fires <- spread(hab, loci = as.integer(sample(1:ncell(hab), 10)),
                returnIndices = TRUE, 0.235, 0, NULL, 1e8, 8, iterations = 3, id = TRUE)
fires[, list(size = length(initialLocus)), by = id]  # See sizes of fires

fires2 <- spread(hab, loci = NA_real_, returnIndices = TRUE, 0.235, 0, NULL,
                 1e8, 8, iterations = 2, id = TRUE, spreadState = fires)
# NOTE events are assigned arbitrary IDs, starting at 1
```

```
## Use data.table and loci...
fires <- spread(hab, loci = as.integer(sample(1:ncell(hab), 10)),
                returnIndices = TRUE, 0.235, 0, NULL, 1e8, 8,
                iterations = 2, id = TRUE)
fullRas <- rast(hab)
fullRas[] <- 1:ncell(hab)
burned <- fires[active == FALSE]
burnedMap <- rasterizeReduced(burned, fullRas, "id", "indices")

terra::plot(burnedMap, type = "classes")

####################
## stopRule examples
####################

# examples with stopRule, which means that the eventual size is driven by the values on the raster
#  passed in to the landscape argument. It won't be exact because the pixel values
#  will likely not allow it
stopRule22 <- function(landscape) sum(landscape) > 100

set.seed(1234)
stopRule1 <- function(landscape) sum(landscape) > 50
stopRuleA <- spread(hab, loci = as.integer(sample(1:ncell(hab), 10)), 1, 0, NULL,
                  maxSize = 1e6, 8, 1e6, id = TRUE, circle = TRUE, stopRule = stopRule1,
                    stopRuleBehavior = "excludePixel")
tapply(hab[], stopRuleA[], sum) # all below 50

set.seed(1234)
# using stopRuleBehavior = "excludePixel"
stopRuleB <- spread(hab, loci = as.integer(sample(1:ncell(hab), 10)), 1, 0, NULL,
                  maxSize = 1e6, 8, 1e6, id = TRUE, circle = TRUE, stopRule = stopRule22,
                    stopRuleBehavior = "excludePixel")
tapply(hab[], stopRuleB[], sum) # all below 100

if (interactive())
  terra::plot(c(stopRuleA, stopRuleB))
# Cellular automata shapes
# Diamonds - can make them with: a boolean raster, directions = 4,
#    stopRule in place, spreadProb = 1
diamonds <- spread(hab > 0, spreadProb = 1, directions = 4, id = TRUE, stopRule = stopRule22)

terra::plot(diamonds)

# Squares - can make them with: a boolean raster, directions = 8,
#    stopRule in place, spreadProb = 1
squares <- spread(hab > 0, spreadProb = 1, directions = 8, id = TRUE, stopRule = stopRule22)
terra::plot(squares)

# Interference shapes - can make them with: a boolean raster, directions = 8,
#    stopRule in place, spreadProb = 1
stopRule2 <- function(landscape) sum(landscape) > 200
squashedDiamonds <- spread(hab > 0, spreadProb = 1,
                           loci = (ncell(hab) - ncol(hab)) / 2 + c(4, -4),
```

```
                            directions = 4, id = TRUE, stopRule = stopRule2)
terra::plot(squashedDiamonds)

# Circles with spreadProb < 1 will give "more" circular shapes, but definitely not circles
stopRule2 <- function(landscape) sum(landscape) > 200
seed <- sample(1e4, 1)
set.seed(seed)

circlish <- spread(hab > 0, spreadProb = 1, iterations = 10,
                   loci = (ncell(hab) - ncol(hab)) / 2 + c(4, -4),
                   directions = 8, id = TRUE, circle = TRUE)#, stopRule = stopRule2)
if (interactive())
  terra::plot(c(circlish))

set.seed(seed)
regularCA <- spread(hab > 0, spreadProb = 1, iterations = 10,
                    loci = (ncell(hab) - ncol(hab)) / 2 + c(4, -4),
                    directions = 8, id = TRUE)#, stopRule = stopRule2)

if (interactive()) # compare to circlish
  terra::plot(regularCA)

####################
# complex stopRule
####################

initialLoci <- sample(seq_len(ncell(hab)), 2)
endSizes <- seq_along(initialLoci) * 200

# Can be a function of landscape, id, and/or any other named
#   variable passed into spread
stopRule3 <- function(landscape, id, endSizes) sum(landscape) > endSizes[id]

set.seed(1)
twoCirclesDiffSize <- spread(hab, spreadProb = 1, loci = initialLoci,
                             circle = TRUE, directions = 8, id = TRUE,
                             stopRule = stopRule3, endSizes = endSizes,
                             stopRuleBehavior = "excludePixel")

# or using named list of named elements:
set.seed(1)
twoCirclesDiffSize2 <- spread(hab, spreadProb = 1, loci = initialLoci,
                              circle = TRUE, directions = 8, id = TRUE,
                              stopRule = stopRule3,
                              vars = list(endSizes = endSizes),
                              stopRuleBehavior = "excludePixel")

compareGeom(twoCirclesDiffSize, twoCirclesDiffSize2, res = TRUE,
            stopOnError = FALSE)

terra::plot(c(twoCirclesDiffSize , twoCirclesDiffSize2))

cirs <- values(twoCirclesDiffSize)
```

```
vals <- tapply(hab[][cirs > 0], cirs[cirs > 0], sum) # one is <200, other is <400 as per endSizes

# Stop if sum of landscape is big or mean of quality is too small
quality <- rast(hab)
quality[] <- runif(ncell(quality), 0, 1)
stopRule4 <- function(landscape, quality, cells) {
  (sum(landscape) > 20) | (mean(values(quality)[cells]) < 0.3)
}

twoCirclesDiffSize <- spread(hab, spreadProb = 1, loci = initialLoci, circle = TRUE,
                             directions = 8, id = TRUE, stopRule = stopRule4,
                             quality = quality, stopRuleBehavior = "excludePixel")


## Using alternative algorithm, not probabilistic diffusion
## Will give exactly correct sizes, yet still with variability
## within the spreading (i.e., cells with and without successes)
seed <- sample(1e6, 1)
set.seed(seed)
startCells <- startCells[1:4]
maxSizes <- rexp(length(startCells), rate = 1 / 500)
fires <- spread(hab, loci = startCells, 1, persistence = 0,
                neighProbs = c(0.5, 0.5, 0.5) / 1.5,
                mask = NULL, maxSize = maxSizes, directions = 8,
                iterations = 1e6, id = TRUE, plot.it = FALSE, exactSizes = TRUE)
all(table(fires[fires > 0][]) == floor(maxSizes))

terra::plot(fires)
hist(fires[][fires[] > 0], main = "fire size distribution")

## Example with relativeSpreadProb ... i.e., a relative probability spreadProb
##  (shown here because because spreadProb raster is not a probability).
##  Here, we force the events to grow, choosing always 2 neighbours,
##  according to the relative probabilities contained on hab layer.
##
## Note: `neighProbs = c(0,1)` forces each active pixel to move to 2 new pixels
## (`prob = 0` for 1 neighbour, `prob = 1` for 2 neighbours)
##
## Note: set hab3 to be very distinct probability differences, to detect spread
##  differences
hab3 <- (hab < 20) * 200 + 1
seed <- 643503
set.seed(seed)
sam <- sample(which(hab3[] == 1), 1)
set.seed(seed)
events1 <- spread(hab3, spreadProb = hab3, loci = sam, directions = 8,
                  neighProbs = c(0, 1), maxSize = c(70), exactSizes = TRUE)

# Compare to absolute probability version
set.seed(seed)
events2 <- spread(hab3, id = TRUE, loci = sam, directions = 8,
                  neighProbs = c(0, 1), maxSize = c(70), exactSizes = TRUE)
```

```
terra::plot(events1)

terra::plot(events2, col = c("white", "red", "red"))

hist(events1[], breaks = 30, main = "Event size distribution") ## TODO: fix this plot
# Compare outputs -- should be more high value hab pixels spread to in event1
#  (randomness may prevent this in all cases)
sum(hab3[events1[] > 0]) >= sum(hab3[events2[] > 0]) ## should be usually TRUE

# clean up
data.table::setDTthreads(origDTThreads)
options(Ncpus = origNcpus)
```

---

spread2 *Simulate a contagious spread process on a landscape, with* `data.table` *internals*

Description

This can be used to simulate fires, seed dispersal, calculation of iterative, concentric, symmetric (currently) landscape values and many other things. Essentially, it starts from a collection of cells (`start`, called "events") and spreads to neighbours, according to the `directions` and `spreadProb` with modifications due to other arguments. **NOTE:** `spread()` is similar, but sometimes slightly faster, but less robust, and more difficult to use iteratively. `spread2` is **not** a drop-in replacement for `spread`: several arguments are named differently, and the returned `data.table` shares no column names with the one returned by `spread(returnIndices = TRUE)` (e.g. `pixels` rather than `indices`, `initialPixels` rather than `initialLocus`, and a character `state` column rather than a logical `active` column). See the section "Differences between `spread` and `spread2`" below before converting existing code.

Usage

```
spread2(
  landscape,
  start = ncell(landscape)/2 - ncol(landscape)/2,
  spreadProb = 0.23,
  persistProb = NA_real_,
  asRaster = TRUE,
  maxSize,
  exactSize,
  directions = 8L,
  iterations = 1000000L,
  returnDistances = FALSE,
  returnDirections = FALSE,
  returnFrom = FALSE,
  maxRetriesPerID = 10,
  spreadProbRel = NA_real_,
  plot.it = FALSE,
```

```
  circle = FALSE,
  asymmetry = NA_real_,
  asymmetryAngle = NA_real_,
  allowOverlap = 0,
  neighProbs = NA_real_,
  oneNeighbourOnly = FALSE,
  skipChecks = FALSE
)
```

Arguments

landscape	Required. A SpatRaster or RasterLayer object. This defines the possible locations for spreading events to start and spread2 into. Required.
start	Required. Either a vector of pixel numbers to initiate spreading, or a data.table that is the output of a previous spread2. If a vector, they should be cell indices (pixels) on the landscape. If user has x and y coordinates, these can be converted with cellFromXY().
spreadProb	Numeric of length 1 or length ncell(landscape) or a SpatRaster or RasterLayer that is the identical dimensions as landscape. If numeric of length 1, then this is the global (absolute) probability of spreading into each cell from a neighbour. If a numeric of length ncell(landscape) or a raster, then this must be the cell-specific (absolute) probability of a "receiving" potential cell. Default is 0.23. If relative probabilities are required, use spreadProbRel. If used together, then the relative probabilities will be re-scaled so that the mean relative probability of potential neighbours is equal to the mean of spreadProb of the potential neighbours.
persistProb	Numeric of length 1, or a SpatRaster or RasterLayer. If numeric of length 1, then this is the global (absolute) probability of cell continuing to burn per time step. If a raster, then this must be the cell-specific (absolute) probability of a fire persisting. Default is NA, which is the same as 0, i.e. a cell only burns for one time step.
asRaster	Logical, length 1. If TRUE, the function will return a Raster where raster non NA values indicate the cells that were "active", and the value is the initial starting pixel.
maxSize	Numeric. Maximum number of cells for a single or all events to be spread2. Recycled to match start length, if it is not as long as start. This will be overridden if exactSize also provided. See section on 'Breaking out of spread2 events'.
exactSize	Numeric vector, length 1 or length(start). Similar to maxSize, but these will be the exact final sizes of the events. i.e., the spread2 events will continue until they are floor(exactSize). This will override maxSize if both provided. See Details.
directions	The number adjacent cells in which to look; default is 8 (Queen case). Can only be 4 or 8.
iterations	Number of iterations to spread2. Leaving this NULL allows the spread2 to continue until stops spreading itself (i.e., exhausts itself).

returnDistances
: Logical. Should the function include a column with the individual cell distances from the locus where that event started. Default is FALSE. See Details.

returnDirections
: Logical. Should the function include a column with the individual directions (in radians) from the locus where that event started. Default is FALSE.

returnFrom
: Logical. Should the function return a column with the source, i.e, the lag 1 "from" pixel, for each iteration.

maxRetriesPerID
: Only active if `exactSize` is used. This is the number of attempts that will be made per event ID, before abandoning, therefore completing the `spread2` for that event with a size that is smaller than `exactSize`. Default 10 times.

spreadProbRel
: Optional `RasterLayer`, `SpatRaster`, or numeric vector of length `ncell(landscape)`, indicating a surface of relative probabilities useful when using `neighProbs` (which provides a mechanism for selecting a specific number of cells at each iteration). This indicates the relative probabilities for the selection of successful neighbours. `spreadProb` will still be evaluated *after* the relative probabilities and `neighProbs` has been evaluated, i.e., potential cells will be identified, then some could be rejected via `spreadProb`. If absolute `spreadProb` is not desired, *be sure to set* `spreadProb = 1`. Ignored if `neighProbs` is not provided.

plot.it
: If `TRUE`, then plot the raster at every iteration, so one can watch the `spread2` event grow.

circle
: Logical. If TRUE, then outward `spread2` will be by equidistant rings, rather than solely by adjacent cells (via `directions` arg.). Default is `FALSE`. Using `circle = TRUE` can be dramatically slower for large problems. Note, this will likely create unexpected results if `spreadProb < 1`.

asymmetry
: A numeric, `SpatRaster` or `RasterLayer` indicating the ratio of the asymmetry to be used. i.e., 1 is no asymmetry; 2 means that the angles in the direction of the `asymmetryAngle` are 2x the `spreadProb` of the angles opposite tot he `asymmetryAngle` Default is NA, indicating no asymmetry. See details. This is still experimental. Use with caution.

asymmetryAngle
: A numeric, `SpatRaster` or `RasterLayer` indicating the angle in degrees (0 is "up", as in North on a map), that describes which way the `asymmetry` is.

allowOverlap
: `numeric` (`logical` will work for backwards compatibility). See details. Default is 0, i.e., no overlapping.

neighProbs
: An optional numeric vector, whose sum is 1. It indicates the probabilities that an individual spread iteration will spread to `1, 2, ..., length(neighProbs)` neighbours, respectively. If this is used (i.e., something other than `NA`), `circle` and `returnDistances` will not work currently.

oneNeighbourOnly
: Logical. Default is `FALSE`. If `TRUE`, then this spread algorithm will allow exactly one neighbour to be spread to (not fewer or more). This could be used, e.g., for an animal moving. If this is `TRUE`, then `allowOverlap` will be set to `2` if it is `0` or `1`.

skipChecks
: Logical. If TRUE, the argument checking (i.e., assertions) will be skipped. This should likely only be used once it is clear that the function arguments are well

understood and function speed is of the primary importance. This is likely most useful in repeated iteration cases i.e., if this call is using the previous output from this same function.

Details

There are 2 main underlying algorithms for active cells to "spread" to nearby cells (adjacent cells): `spreadProb` and `neighProb`. Using `spreadProb`, every "active" pixel will assess all neighbours (either 4 or 8, depending on `directions`), and will "activate" whichever neighbours successfully pass independent calls to `runif(1,0,1) < spreadProb`. The algorithm will iterate again and again, each time starting from the newly "activated" cells. Several built-in decisions are as follows:

1. no active cell can activate a cell that was already activated by the same event (i.e., "it won't go backwards");
2. If `allowOverlap` is `FALSE`, then the previous rule will also apply, regardless of which "event" caused the pixels to be previously active.

This function can be interrupted before all active cells are exhausted if the `iterations` value is reached before there are no more active cells to `spread2` into. The interrupted output (a `data.table`) can be passed subsequently as an input to this same function (as `start`). This is intended to be used for situations where external events happen during a `spread2` event, or where one or more arguments to the `spread2` function change before a `spread2` event is completed. For example, if it is desired that the `spreadProb` change before a `spread2` event is completed because, for example, a fire is spreading, and a new set of conditions arise due to a change in weather.

`asymmetry` here is slightly different than in the `spread` function, so that it can deal with a `RasterLayer` of `asymmetryAngle`. Here, the `spreadProb` values of a given set of neighbours around each active pixel are adjusted to create `adjustedSpreadProb` which is calculated maintain the following two qualities:

$$mean(spreadProb) = mean(ajustedSpreadProb)$$

and

$$max(spreadProb)/min(spreadProb) = asymmetry$$

along the axis of `asymmetryAngle`. NOTE: this means that the 8 neighbours around an active cell may not fulfill the preceeding equality if `asymmetryAngle` is not exactly one of the 8 angles of the 8 neighbours. This means that

$$max(spreadProb)/min(spreadProb)$$

will generally be less than `asymmetry`, for the 8 neighbours. The exact adjustment to the spreadProb is calculated with:

$$angleQuality < -(cos(angles - rad2(asymmetryAngle)) + 1)/2$$

which is multiplied to get an angle-adjusted spreadProb:

$$spreadProbAdj < -actualSpreadProb * angleQuality$$

which is then rescaled:

$$adjustedSpreadProb = (spreadProbAdj - min(spreadProbAdj)) * par2 + par1$$

, where par1 and par2 are parameters calculated internally to make the 2 conditions above true.

If maxSize or exactSize are used, then spreading will continue and stop before or at maxSize or at exactSize, respectively. If iterations is specified, then the function will end, and the returned data.table may (if maxSize) or will (if exactSize) have at least one active cell per event that did not already achieve maxSize or exactSize. This will be very useful to build new, customized higher-level wrapper functions that iteratively call spread2.

Value

Either a data.table (asRaster=FALSE) or a raster of the same class as landscape – a SpatRaster, or a RasterLayer if that is what was passed in (asRaster=TRUE, the default). The data.table will have one attribute named spreadState, which is a list containing a data.table of current cluster-level information about the spread events. If asRaster=TRUE, then the data.table (with its spreadState attribute) will be attached to the Raster as an attribute named pixel as it provides pixel-level information about the spread events.

That raster represents every cell in which a successful spread2 event occurred. For the case of, say, a fire this would represent every cell that burned. If allowOverlap is TRUE, the return will always be a data.table.

If asRaster is FALSE, then this function returns a data.table with 3 (or 4 if returnFrom is TRUE) columns:

initialPixels	the initial cell number of that particular spread2 event.
pixels	The cell indices of cells that have been touched by the spread2 algorithm.
state	a logical indicating whether the cell is active (i.e., could still be a source for spreading) or not (no spreading
from	The pixel indices that were the immediately preceding "source" for each pixels, i.e., the lag 1 pixels. Only

The attribute saved with the name "spreadState" (e.g., attr(output, "spreadState")) includes a data.table with columns:

id	An arbitrary code, from 1 to length(start) for each "event".
initialPixels	the initial cell number of that particular spread2 event.
numRetries	The number of re-starts the event did because it got stuck (normally only because exactSize was used and
maxSize	The number of pixels that were provided as inputs via maxSize or exactSize.
size	The current size, in pixels, of each event.

and several other objects that provide significant speed ups in iterative calls to spread2. If the user runs spread2 iteratively, there will likely be significant speed gains if the data.table passed in to start should have the attribute attached, or re-attached if it was lost, e.g., via setattr(outInput, "spreadState", attr(out, "spreadState")), where out is the returned data.table from the previous call to spread2, and outInput is the modified data.table. Currently, the modified data.table **must** have the same order as out.

Breaking out of spread2 events

There are 3 ways for the spread2 to "stop" spreading. Here, each "event" is defined as all cells that are spawned from each unique start location. The ways outlined below are all acting at all

times, i.e., they are not mutually exclusive. Therefore, it is the user's responsibility to make sure the different rules are interacting with each other correctly.

spreadProb	Probabilistically, if spreadProb is low enough, active spreading events will stop. In practice, this number gene
maxSize	This is the number of cells that are "successfully" turned on during a spreading event. spreadProb will still be
exactSize	This is the number of cells that are "successfully" turned on during a spreading event. This will override an eve
iterations	This is a hard cap on the number of internal iterations to complete before returning the current state of the syste

Building custom spreading events

This function can be used iteratively, with relatively little overhead compared to using it non-iteratively. In general, this function can be called with arguments set as user needs, and with specifying e.g., `iterations = 1`. This means that the function will spread outwards 1 iteration, then stop. The returned object will be a `data.table` or raster that can be passed immediately back as the start argument into a subsequent call to `spread2`. This means that every argument can be updated at each iteration.

When using this function iteratively, there are several things to keep in mind. The output will likely be sorted differently than the input (i.e., the order of start, if a vector, may not be the same order as that returned). This means that when passing the same object back into the next iteration of the function call, `maxSize` or `exactSize` may not be in the same order. To get the same order, the easiest thing to do is sort the initial `start` objects by their pixel location, increasing. Then, of course, sorting any vectorized arguments (e.g., `maxSize`) accordingly.

NOTE: the returned `data.table` or raster should not be altered when passed back into `spread2`.

`allowOverlap`

If `1` (or `TRUE`), then individual events can overlap with one another, i.e., allow overlap between events. If `2` (or `NA`), then each pixel is essentially independent, allowing overlap between and within events. This likely requires a user to intervene as it is possible to spread back onto itself. If `3` (did not exist previously), individual events can overlap, and there can be overlap within an event, but only within an iteration, i.e., once an iteration is finished, and a pixel was activated, then the spreading will not return onto these pixels. If `0` (or `FALSE`), then once a pixel is activated, it cannot be re-activated, within or between event. This allows events to not interfere with one another i.e., they do not interact (this is slower than if `allowOverlap = FALSE`). Default is 0. In the case of 2 or 3, this would be, perhaps, useful for dispersal of, say, insect swarms.

Using `spreadCpp()` instead

With `options(spades.useSpreadCpp = TRUE)`, a call to this function is handed to the faster `spreadCpp()`, with its arguments and return value converted, so existing code runs unchanged. Results differ from this function's for the same seed (see `spreadCpp()`). A call using an argument `spreadCpp()` does not support runs this function as usual, with a message once per session. `exactSize` becomes `minSize = maxSize` with `minSizeTries = 0`, and `jumpTries = maxRetriesPerID %/% 10`. `spreadCpp()` returns a finished spread, every cell `"inactive"` and no `"spreadState"` attribute, so a call with `iterations` below its default, which stops part-way to be resumed, runs this function. As in this function, an `NA` in `maxSize` or `exactSize` means no size limit for any event, and sizes are rounded down to whole cells.

Differences between `spread` and `spread2`

Which to use. `spread2()` is the more robust of the two: its behaviour is better tested, it can be called iteratively (passing its own output back in as `start`), and more combinations of its arguments work as documented. `spread()` is generally faster, sometimes substantially so, but several of its argument combinations are untested, deprecated, or unimplemented.

Argument correspondence. Arguments with the same name in both functions generally mean the same thing, with the exceptions noted below.

`spread`	`spread2`
`loci`	`start`
`persistence`	`persistProb`
`returnIndices`	`asRaster`
`exactSizes`	`exactSize`
`spreadState`	`start`
`quick`, `skipChecks`	`skipChecks`
`relativeSpreadProb`	`spreadProbRel`
`allowOverlap`	`allowOverlap`
`maxSize`	`maxSize`
`mask`, `mapID`, `lowMemory`	–
`id`, `stopRule`, `stopRuleBehavior`, `circleMaxRadius`	–
–	`returnDirections`, `returnFrom`, `oneNeighbourOnly`, `maxRetries`

Output column names. This is the most common source of errors when converting code. The returned `data.tables` share *no* column names:

Meaning	`spread(returnIndices = TRUE)`	`spread2(asRaster = FALSE)`
Cell touched by the spread	`indices`	`pixels`
Cell the event started from	`initialLocus`	`initialPixels`
Whether the cell can still spread	`active` (logical)	`state` (character: `"activeSource"`, `"success`
Event identifier	`id`	not in the main table; see `attr(out, "spread`
Distance from origin	`dists` (only with `circle`/`allowOverlap`)	`distance` (with `returnDistances`)
Source of the previous step	–	`from` (with `returnFrom`)

Note

`exactSize` may not be achieved if there aren't enough cells in the map. Also, `exactSize` may not be achieved because the active cells are "stuck", i.e., they have no inactivated cells to move to; or the `spreadProb` is low. In the latter two cases, the algorithm will retry again, but it will only retry from the last iteration's active cells. The algorithm will only retry 10 times before quitting. Currently, there will also be an attempt to "jump" up to four cells away from the active cells to try to continue spreading.

A common way to use this function is to build wrappers around this, followed by iterative calls in a `while` loop. See example.

Author(s)

Eliot McIntire and Steve Cumming

See Also

spread() for a different implementation of the same algorithm. spread is less robust but it is often slightly faster.

Examples

```
library(terra)

origDTThreads <- data.table::setDTthreads(2L)
origNcpus <- options(Ncpus = 2L)

a <- rast(ext(0, 10, 0, 10), res = 1)
sams <- sort(sample(ncell(a), 3))

# Simple use -- similar to spread(...)
out <- spread2(a, start = sams, 0.225)
if (interactive()) {
  terra::plot(out)
}

# Use maxSize -- this gives an upper limit
maxSizes <- sort(sample(1:10, size = length(sams)))
out <- spread2(a, start = sams, 0.225, maxSize = maxSizes, asRaster = FALSE)
# check TRUE using data.table .N
out[, .N, by = "initialPixels"]$N <= maxSizes

# Use exactSize -- gives an exact size, if there is enough space on the Raster
exactSizes <- maxSizes
out <- spread2(a, start = sams, spreadProb = 0.225,
               exactSize = exactSizes, asRaster = FALSE)
out[, .N, by = "initialPixels"]$N == maxSizes # should be TRUE TRUE TRUE

# Use exactSize -- but where it can't be achieved
exactSizes <- sort(sample(100:110, size = length(sams)))
out <- spread2(a, start = sams, 1, exactSize = exactSizes)

# Iterative calling -- create a function with a high escape probability
spreadWithEscape <- function(ras, start, escapeProb, spreadProb) {
  out <- spread2(ras, start = sams, spreadProb = escapeProb, asRaster = FALSE)
  while (any(out$state == "sourceActive")) {
    # pass in previous output as start
    out <- spread2(ras, start = out, spreadProb = spreadProb,
                   asRaster = FALSE, skipChecks = TRUE) # skipChecks for speed
  }
  out
}

set.seed(421)
out1 <- spreadWithEscape(a, sams, escapeProb = 0.25, spreadProb = 0.225)
set.seed(421)
out2 <- spread2(a, sams, 0.225, asRaster = FALSE)
# The one with high escape probability is larger (most of the time)
```

```
NROW(out1) > NROW(out2) ## TODO: not true

## Use neighProbs, with a spreadProb that is a RasterLayer
# Create a raster of different values, which will be the relative probabilities
#   i.e., they are rescaled to relative probabilities within the 8 neighbour choices.
#   The neighProbs below means 70% of the time, 1 neighbour will be chosen,
#   30% of the time 2 neighbours.
#   The cells with spreadProb of 5 are 5 times more likely than cells with 1 to be chosen,
#   when they are both within the 8 neighbours
sp <- rast(ext(0, 3, 0, 3), res = 1, vals = 1:9) #small raster, simple values
# Check neighProbs worked
out <- list()

# enough replicates to see stabilized probabilities
for (i in 1:100) {
  out[[i]] <- spread2(sp, spreadProbRel = sp, spreadProb = 1,
                      start = 5, iterations = 1,
                      neighProbs = c(1), asRaster = FALSE)
}
out <- data.table::rbindlist(out)[pixels != 5] # remove starting cell
table(sp[out$pixels])
# should be non-significant -- note no 5 because that was the starting cell
#  This tests whether the null model is true ... there should be proportions
#  equivalent to 1:2:3:4:6:7:8:9 ... i.e,. cell 9 should have 9x as many events
#  spread to it as cell 1. This comes from sp object above which is providing
#  the relative spread probabilities
keep <- c(1:4, 6:9)
chisq.test(keep, unname(tabulate(sp[out$pixels]$lyr.1, 9)[keep]),
           simulate.p.value = TRUE)

## Example showing asymmetry
sams <- ncell(a) / 4 - ncol(a) / 4 * 3
circs <- spread2(a, spreadProb = 0.213, start = sams,
                 asymmetry = 2, asymmetryAngle = 135,
                 asRaster = TRUE)
## ADVANCED: Estimate spreadProb when using asymmetry, such that the expected
##   event size is the same as without using asymmetry

  if (interactive()) {
   # Still using `raster` as it works more easily with parallelism due to not using pointers
    #   This will updated at a later release
    if (requireNamespace("raster", quietly = TRUE)) {
      ras <- raster::raster(a)
      ras[] <- 1

      n <- 100
      sizes <- integer(n)
      for (i in 1:n) {
        circs <- spread2(ras, spreadProb = 0.225,
                         start = round(ncell(ras) / 4 - ncol(ras) / 4 * 3),
                         asRaster = FALSE)
        sizes[i] <- circs[, .N]
      }
```

```
goalSize <- mean(sizes)

if (requireNamespace("DEoptim", quietly = TRUE)) {
  library(parallel)
  library(DEoptim)

  # need 10 cores for 10 populations in DEoptim
  cl <- makeCluster(pmin(10, detectCores() - 2))
  parallel::clusterEvalQ(cl, {
    library(SpaDES.tools)
    library(terra)
    library(raster)
    library(fpCompare)
  })

  objFn <- function(sp, n = 20, ras, goalSize) {
    sizes <- integer(n)
    for (i in 1:n) {
     circs <- spread2(ras, spreadProb = sp, start = ncell(ras) / 4 - ncol(ras) / 4 * 3,
                      asymmetry = 2, asymmetryAngle = 135,
                      asRaster = FALSE)
      sizes[i] <- circs[, .N]
    }
    abs(mean(sizes) - goalSize)
  }
  aa <- DEoptim(objFn, lower = 0.2, upper = 0.23,
                control =
                  DEoptim.control(
                    cluster = cl, NP = 10, VTR = 0.02,
                   # imposing itermax simply for example; should let go to completion
                    itermax = 5,
                    initialpop = as.matrix(rnorm(10, 0.213, 0.001))),
                ras = ras, goalSize = goalSize)

  # The value of spreadProb that will give the
  #    same expected event sizes to spreadProb = 0.225 is:
  sp <- aa$optim$bestmem
 circs <- spread2(ras, spreadProb = sp, start = ncell(ras) / 4 - ncol(ras) / 4 * 3,
                  asymmetry = 2, asymmetryAngle = 135, asRaster = FALSE)

  stopCluster(cl)
 }
}

# clean up
data.table::setDTthreads(origDTThreads)
options(Ncpus = origNcpus)
```

---

spread3 *An alternative spread function, conceived for insects*

Description

This is built with `spread2()` and is still experimental. This one differs from other attempts in that it treats the advection and dispersal as mathematical vectors that are added together. They are "rounded" to pixel centres.

Usage

```
spread3(
  start,
  rasQuality,
  rasAbundance,
  advectionDir,
  advectionMag,
  meanDist,
  dispersalKernel = "exponential",
  sdDist = 1,
  plot.it = 2,
  minNumAgents = 50,
  verbose = getOption("LandR.verbose", 0),
  saveStack = NULL,
  skipChecks = FALSE
)
```

Arguments

`start`	Raster indices from which to initiate dispersal
`rasQuality`	A raster with habitat quality. Currently, must be scaled from 0 to 1, i.e., a probability of "settling"
`rasAbundance`	A raster where each pixel represents the number of "agents" or pseudo-agents contained. This number of agents, will be spread horizontally, and distributed from each pixel that contains a non-zero non NA value.
`advectionDir`	A single number or `RasterLayer` in degrees from North = 0 (though it will use radians if all values are `abs(advectionDir) > 2 * pi`). This indicates the direction of advective forcing (i.e., wind).
`advectionMag`	A single number or `RasterLayer` in distance units of the `rasQuality`, e.g., meters, indicating the relative forcing that will occur. It is imposed on the total event, i.e., if the `meanDist` is `10000`, and `advectionMag` is `5000`, then the expected distance (i.e., `63%` of agents) will have settled by `15000` map units.
`meanDist`	A single number indicating the mean distance parameter in map units (not pixels), for a negative exponential distribution dispersal kernel (e.g., `dexp`). This will mean that `63%` of agents will have settled at this `meanDist` (still experimental).
`dispersalKernel`	One of either `"exponential"` or `"weibull"`.
`sdDist`	A single number indicating the `sd` parameter of a two-parameter `dispersalKernel`. Defaults to `1`, which is the same as the `exponential` distribution.

plot.it	Numeric. With increasing numbers above 0, there will be plots produced during iterations. Currently, only 0, 1, or 2+ are distinct.
minNumAgents	Single numeric indicating the minimum number of agents to consider all dispersing finished. Default is 50.
verbose	Numeric. With increasing numbers above 0, there will be more messages produced. Currently, only 0, 1, or 2+ are distinct.
saveStack	If provided as a character string, it will save each iteration as part of a rasterStack to disk upon exit.
skipChecks	Logical. If TRUE, assertions will be skipped (faster, but could miss problems)

Value

A data.table with all information used during the spreading

Examples

```
## these tests are fairly heavy, so don't run during automated tests
############################################################
# Simple case, no variation in rasQuality, numeric advectionDir and advectionMag
############################################################

  library(terra)

  origDTThreads <- data.table::setDTthreads(2L)
  origNcpus <- options(Ncpus = 2L)

  maxDim <- 10000
  ras <- terra::rast(terra::ext(c(0, maxDim, 0, maxDim)), res = 100, vals = 0)
  rasQuality <- terra::rast(ras)
  rasQuality[] <- 1
  rasAbundance <- terra::rast(rasQuality)
  rasAbundance[] <- 0
  # startPixel <- middlePixel(rasAbundance)
  startPixel <- sample(seq(terra::ncell(rasAbundance)), 30)
  rasAbundance[startPixel] <- 1000
  advectionDir <- 70
  advectionMag <- 4 * res(rasAbundance)[1]
  meanDist <- 2600

  # Test the dispersal kernel -- create a function
  plotDispersalKernel <- function(out, meanAdvectionMag) {
    out[, disGroup := round(distance / 100) * 100]
    freqs <- out[, .N, by = "disGroup"]
    freqs[, `:=`(cumSum = cumsum(N), N = N)]
    plot(freqs$disGroup, freqs$cumSum, # addTo = "CumulativeNumberSettled",
         main = "Cumulative Number Settled") # can plot the distance X number
    abline(v = meanAdvectionMag + meanDist)
    newTitle <- "Number Settled By Distance"
    plot(freqs$disGroup, freqs$N, # addTo = gsub(" ", "", newTitle),
         main = newTitle) # can plot the distance X number
    abline(v = meanAdvectionMag + meanDist)
```

```
  # should be 0.63:
  freqs[disGroup == meanAdvectionMag + meanDist, cumSum] / tail(freqs, 1)[, cumSum]
  mtext(side = 3, paste("Average habitat quality: ",
                        round(mean(rasQuality[], na.rm = TRUE), 2)),
        outer = TRUE, line = -2, cex = 2)
}
out <- spread3(rasAbundance = rasAbundance,
               rasQuality = rasQuality,
               advectionDir = advectionDir,
               advectionMag = advectionMag,
               meanDist = meanDist, verbose = 2,
               plot.it = interactive())

plotDispersalKernel(out, advectionMag)

# The next examples are potentially time consuming; avoid on automated testing
if (interactive()) {
  ##########################################################
  ### The case of variable quality raster
  ##########################################################
  rasQuality <- terra::rast(system.file("extdata", "rasQuality.tif",
                                        package = "SpaDES.tools"))
 terra::crs(rasQuality) <- system.file("extdata", "targetCRS.rds", package = "SpaDES.tools") |>
    readRDS() |>
    slot("projargs")
  mask <- rasQuality < 5
  rasQuality[mask[] %in% TRUE] <- 0
  # rescale so min is 0.75 and max is 1
  rasQuality[] <- rasQuality[] / (reproducible::maxFn(rasQuality) * 4) + 1 / 4
  rasAbundance <- terra::rast(rasQuality)
  rasAbundance[] <- 0
  startPixel <- sample(seq(ncell(rasAbundance)), 300)
  rasAbundance[startPixel] <- 1000
  advectionDir <- 75
  advectionMag <- 4 * res(rasAbundance)[1]
  meanDist <- 2600
  out <- spread3(rasAbundance = rasAbundance,
                 rasQuality = rasQuality,
                 advectionDir = advectionDir,
                 advectionMag = advectionMag,
                 meanDist = meanDist, verbose = 2,
                 plot.it = interactive())
  if (interactive()) {
    plotDispersalKernel(out, advectionMag)
  }

  ##############################################################################
  ### The case of variable quality raster, raster for advectionDir & advectionMag
  ##############################################################################
  maxDim <- 10000
  ras <- terra::rast(terra::ext(c(0, maxDim, 0, maxDim)), res = 100, vals = 0)
  rasQuality <- terra::rast(ras)
  rasQuality[] <- 1
```

```
rasAbundance <- terra::rast(rasQuality)
rasAbundance[] <- NA
# startPixel <- middlePixel(rasAbundance)
startPixel <- sample(seq(ncell(rasAbundance)), 25)
rasAbundance[startPixel] <- 1000

# raster for advectionDir
advectionDir <- terra::rast(system.file("extdata", "advectionDir.tif",
                                        package = "SpaDES.tools"))
crs(advectionDir) <- crs(rasQuality)
# rescale so min is 0.75 and max is 1
advectionDir[] <- advectionDir[] / (reproducible::maxFn(advectionDir)) * 180

# raster for advectionMag
advectionMag <- terra::rast(system.file("extdata", "advectionMag.tif",
                                        package = "SpaDES.tools"))
crs(advectionMag) <- crs(rasQuality)
# rescale so min is 0.75 and max is 1
advectionMag[] <- advectionMag[] / (reproducible::maxFn(advectionMag)) * 600

out <- spread3(rasAbundance = rasAbundance,
               rasQuality = rasQuality,
               advectionDir = advectionDir,
               advectionMag = advectionMag,
               meanDist = meanDist, verbose = 2,
               plot.it = interactive())

if (interactive()) {
  names(advectionDir) <- "Wind direction"
  names(advectionMag) <- "Wind speed"
  names(rasAbundance) <- "Initial abundances"
  terra::plot(c(advectionDir, advectionMag, rasAbundance))

  plotDispersalKernel(out, mean(advectionMag[]))
}

########################################
# save iterations to a stack to make animated GIF
########################################
tmpStack <- tempfile(pattern = "stackToAnimate", fileext = ".tif")
out <- spread3(rasAbundance = rasAbundance,
               rasQuality = rasQuality,
               advectionDir = advectionDir,
               advectionMag = advectionMag,
               meanDist = 2600, verbose = 2,
               plot.it = interactive(), saveStack = tmpStack)

## This animates the series of images into an animated GIF
if (require(animation, quietly = TRUE)) {
  out2 <- terra::rast(tmpStack)
  gifName <- file.path(tempdir(), "animation.gif")

  # Only works on some systems; may need to configure
```

```
    # Works on Windows without system adjustments
    if (identical(.Platform$OS.type, "windows"))
      saveGIF(interval = 0.1, movie.name = gifName, expr = {
        for (i in seq(length(names(out2)))) terra::plot(out2[[i]])
      })
  }
}

# clean up
data.table::setDTthreads(origDTThreads)
options(Ncpus = origNcpus)
```

---

spreadCpp *Stochastic outward spread, implemented in C++*

Description

A fast, deliberately narrow alternative to spread() for the common case: a per-cell probability of being spread to, one or more starting cells, an optional per-fire maximum size, and cell indices back.

Usage

```
spreadCpp(
  landscape,
  loci,
  spreadProb,
  maxSize = Inf,
  directions = 8L,
  iterations = Inf,
  minSize = 0,
  jumpTries = 0L,
  jumpMeanDist = 3,
  minSizeTries = 100L,
  stopCells = NULL,
  stopEvent = NULL,
  stopAt = NULL
)
```

Arguments

landscape	A SpatRaster; only its geometry (number of columns and cells) is used.
loci	Integer vector of starting cell indices, one per fire.
spreadProb	Numeric of length 1 or terra::ncell(landscape), or a SpatRaster or RasterLayer congruent with landscape: the probability that a cell is spread to.
maxSize	Numeric of length 1 or length(loci): the largest number of cells each fire may reach. Inf (the default) means no limit.

directions — 4 or 8. Default 8.

iterations — Maximum number of generations. Default Inf.

minSize — Numeric of length 1 or length(loci): a size each fire reaches before the normal generation rule applies. While a fire has fewer than minSize cells, all of its burning cells stay active from one generation to the next (persistence) and draw again, with the same spreadProb, against their still-unburned neighbours, so the fire grows into a patch shaped by spreadProb rather than dying out early. The generation that reaches minSize stops there, exactly. The next generation spreads from every cell of the patch that still has an unburned, burnable neighbour (its perimeter), not only from the cells that caught last; after that only the cells that caught in the last generation spread, as usual. A fire under minSize with no unburned neighbour whose spreadProb is above 0 (and not NA) stops where it is; otherwise only iterations limits how long a very low spreadProb takes to get there. Must not exceed maxSize. The default, 0, is the behaviour without it, with the same random draws. Used to start every escaped fire at the escape size, e.g. minSize = 9 cells for 50 ha at 5.76-ha cells.

jumpTries — Integer, default 0 (off). Only for a fire that is still under minSize and stuck – none of its burning cells has an unburned neighbour with spreadProb above 0 – up to jumpTries attempts are made to jump: a random cell of the fire, a distance drawn from an exponential with mean jumpMeanDist, truncated to 1.5-20 cells so it clears the adjacent ring, and a uniform direction. A target off the landscape, already burned or unburnable is rejected; otherwise it catches with its spreadProb. The first success joins the fire and the fire keeps growing from all its cells. If every attempt fails the fire stops where it is. With 0 no extra random draws are made. This is a fixed rule for fires trapped in small patches, not a fitted spotting process; spotting tied to fire weather is future work.

jumpMeanDist — Mean jump distance in cells (see jumpTries). Default 3.

minSizeTries — Integer, default 100. With minSize above 1, minSize is reached by rejection rather than persistence: each fire spreads normally from its one ignition cell, and a fire that dies below minSize is undone (its cells freed, its rows dropped) and started again from its ignition cell, up to minSizeTries times. A fire that reaches minSize is kept and goes on uncut, so its size, shape and burning front are those of a fire that got there under its own spreadProb – a sample from the fires that reach minSize. After minSizeTries rejections the fire falls back to the persistence rule of minSize (and jumpTries) from its ignition cell. The expected number of tries per fire is 1 / P(reaching minSize), so a low spreadProb or a small burnable patch costs more; the result carries an attribute "minSizeTries", a data.table with each fire's id, its rejected tries, and whether it used the fallback. minSizeTries = 0 gives persistence alone (the behaviour before this argument existed, with the same random draws). Without minSize it has no effect.

stopCells, stopEvent, stopAt — Optional per-fire stop rule; all NULL (the default) leaves the results and the random numbers used exactly as without them. stopCells is a vector of cell indices (indexed like loci), stopEvent the same length: the fire (an index into loci, 1-based) each stop cell belongs to, and stopAt an integer of length 1 or length(loci): the number of a fire's own stop cells at which it stops. A cell

may be a stop cell of several fires, and repeated (cell, fire) pairs count once. Only a fire's own stop cells count towards its `stopAt`, and a fire with no stop cells never stops by this rule. The fire stops the moment it burns its `stopAt`-th stop cell, in the middle of a generation, in the same way as at `maxSize`: the cells already burned stay burned and the other fires carry on. It interacts with `minSize` and `jumpTries` as `maxSize` does: a fire that has stopped is never pushed on to `minSize` or made to jump, and a try rejected under `minSizeTries` is undone with its count. The ignition cell counts if it is a stop cell. Giving `stopCells` needs `stopEvent` and `stopAt`.

Value

A `data.table` keyed on `id`, with integer columns `id`, `initialLocus` and `indices`, and a logical `active` (always `FALSE`, since the spread has finished) – the same shape `spread()` returns for `returnIndices = TRUE`. When `minSize` is above 1 and `minSizeTries` above 0, it also carries an attribute `"minSizeTries"`: a `data.table` with one row per fire (in `loci` order) and columns `id`, `tries` (its rejected attempts) and `fallback` (`TRUE` if it ran out of tries and used persistence). The attribute is transient: most `data.table` operations drop it, e.g. subsetting or `dt[, .N, by = "id"]`, so read it from the returned object straight away, e.g. `attr(out, "minSizeTries")`.

The rules it follows

- **Growth happens in generations.** The cells that caught in one generation are the only ones that try to spread in the next, so fires grow outwards and a fire's extent is tied to how many generations it has had.
- **One draw per (burning cell, unburned neighbour) pair**, compared against `spreadProb` **of the neighbour**, not of the cell doing the spreading. A cell with `k` burning neighbours therefore catches with probability `1 - (1 - p)^k` in that generation, exactly as in `spread()`.
- **A cell belongs to exactly one fire, and is never spread to twice.** Once a cell has caught it is never reconsidered. Burning cells are visited in a random order within a generation, so when two fires reach the same cell neither is systematically favoured.
- **A fire stops at its own** `maxSize` **and never exceeds it.** `spread()` adds a whole generation and then randomly drops the excess; `spreadCpp()` stops on the boundary, which lands on `maxSize` exactly.
- `NA` **in** `spreadProb` **means unburnable**, as in `spread()`.

Limitations

`spreadCpp()` is deliberately narrow. It covers one case – a per-cell probability of being spread to, one or more ignitions, an optional per-fire size cap – and takes only the arguments listed above. Every other argument `spread()` accepts is **unsupported, and passing one is an error rather than being silently ignored**:

`allowOverlap` Not supported. Fires never overlap; a cell belongs to one fire.

`returnDistances`**,** `circle`**,** `circleMaxRadius` Not supported. No distances are computed and spread is not constrained to a disc.

`asymmetry`**,** `asymmetryAngle` Not supported. Spread is isotropic; there is no directional bias.

`neighProbs`, `relativeSpreadProb` Not supported. Every eligible neighbour is drawn for independently; the number of neighbours a cell spreads to is not itself drawn from a distribution, and probabilities are absolute rather than rescaled within a cell's neighbourhood.

`stopRule`, `stopRuleBehavior` Not supported. The only stopping conditions are `maxSize`, `iterations`, and having nowhere left to go.

`exactSizes` Not supported, though `maxSize` is exact in the sense that a fire which reaches its cap stops there.

`persistence` Not supported. A burned cell stays burned and does not re-burn.

`mask` Not supported. Use `NA` (or 0) in `spreadProb` to make cells unburnable.

`spreadState` Not supported. A spread cannot be continued from an earlier one; each call starts from `loci`.

`spreadProbLater` Not supported. One `spreadProb` applies to every generation.

`torus` Not supported. The landscape does not wrap; spread stops at the edges.

`plot.it`, `id`, `returnIndices` Not supported as options. Nothing is plotted, and the return value is always the indices form.

Use `spread()` when any of these are needed.

Reproducibility

`spreadCpp()` is **not** a drop-in reimplementation of `spread()`. It makes its own random draws, so for a given seed it does not reproduce `spread()`'s cells, and swapping one for the other changes results and invalidates cached output. It is reproducible in the ordinary sense: the same seed and the same inputs give the same answer.

What the two do agree on is behaviour in aggregate. Over 300 seeds on a 1.44M-cell landscape the two produce the same total burned area (Wilcoxon `p = 0.34`) and the same fire-size distribution (Kolmogorov-Smirnov `p = 0.56`), with quantiles within 7% of each other from the 10th percentile to the 99th.

Performance

Roughly 2x faster than `spread()` across a range of fire regimes: 1.6-2.4x over nine scenarios spanning 596 to 188,762 burned cells and landscapes from 0.36M to 9M cells, median 2.1x.

The ratio understates the benefit on the expensive calls, so judge it in seconds. Across those nine scenarios the speedup RATIO falls slightly as more burns (correlation -0.37 with burned cells) while the seconds saved rise steeply (correlation +0.95). The scenario with the lowest ratio, 161,701 burned cells at 1.58x, saves 0.048 s per call – 24 times more than the 2.00x scenario at the other end, which saves 0.002 s. On real fire-model fits the same holds: two landscapes whose ratios differ by 30% (1.44x and 1.88x) save almost identical wall time per objective evaluation (9.5 s and 9.6 s).

The gain depends on what fraction of the landscape burns, and it can vanish. The per-cell state is allocated once per call and costs the same in both implementations, so it is pure overhead that neither avoids; what `spreadCpp()` makes cheaper is the spreading itself. When almost nothing burns, that overhead dominates and there is nothing left to win. Measured on one landscape of ~8.4M cells: with 0.13% of it burned, `spreadCpp()` was no faster at all (0.95x), where scenarios burning 0.7% and 4.2% of their landscapes gave 2.1x and 2.4x.

The same allocation makes cost grow with `landscape` size even when the fires do not: holding the fires fixed, going from 1.44M to 9M cells costs an extra 0.035 s per call. So if the fires occupy a small part of a large landscape, crop first and map the indices back – that both speeds up the call and restores the advantage. Cropping 1.44M to 300k cells is worth about 30%.

See Also

`spread()`, which is slower but far more general.

Examples

```
if (requireNamespace("terra", quietly = TRUE)) {
  r <- terra::rast(terra::ext(0, 100, 0, 100), resolution = 1, vals = 0)
  set.seed(1)
  out <- spreadCpp(r, loci = c(2500L, 7500L), spreadProb = 0.23,
                   maxSize = c(50, 100))
  out[, .N, by = "id"]
}
```

`testEquivalentMetadata` *Test that metadata of 2 or more objects is the same*

Description

Currently, only Raster class has a useful method. Defaults to `all(sapply(list(...)[-1], function(x) identical(list(...)[1], x)))`

Usage

```
testEquivalentMetadata(...)
```

Arguments

`...`	2 or more of the same type of object to test for equivalent metadata.

Value

Invisible `NULL`. Called for its side effect: it signals an error if the objects' metadata are not equivalent, and returns silently if they are.

transitions SELES - *Transitioning to next time step (defunct)*

Description

This function was never fully implemented and is now defunct: calling it signals an error.

It was intended to describe the probability of an agent successfully persisting until the next time step, as a SELES-like function maintaining conceptual backwards compatibility with SELES. The implementation it carried could not work – it assigned into `sp::coordinates(agent)`, which errors on any `Spatial*` object whose coordinates are already set, i.e. all of them – so no caller can have depended on its behaviour.

There is no replacement. If you have a use for this, please contact the package maintainer; it can be implemented properly if there is interest.

Usage

```
transitions(p, agent)
```

Arguments

p	realized probability of persisting (i.e., either 0 or 1).
agent	`SpatialPoints*` object.

Value

None. Signals an error.

Author(s)

Eliot McIntire

wrapTorus *Wrap coordinates or pixels in a torus-like fashion*

Description

Generally useful for model development purposes. Primarily used internally in e.g., `crw` if `torus = TRUE`.

Usage

```
wrapTorus(X, bounds, withHeading = FALSE)

wrap(X, bounds, withHeading = FALSE)
```

Arguments

X	SpatVector, sf, or matrix of coordinates.
bounds	Either a SpatRaster*, Extent, or bbox object defining bounds to wrap around.
withHeading	logical. If TRUE, the previous points must be wrapped also so that the subsequent heading calculation will work. Default FALSE. See details.

Details

This function was named wrap() until version 2.1.3. It was renamed because terra::wrap() means something entirely different – it serialises a SpatRaster or SpatVector so it can be saved or sent to a parallel worker. Since this package imports terra, both were routinely on the search path and whichever was attached last silently masked the other, so a call to wrap() could do either thing depending on load order. wrap() remains as a deprecated alias.

If withHeading used, then X must be an sf or SpatVector object that contains two columns, x1 and y1, with the immediately previous agent locations.

wrap() is deprecated in favour of wrapTorus(); see the note above on why it was renamed.

Value

Object of the same class as X, but with coordinates updated to reflect the wrapping.

Author(s)

Eliot McIntire

Examples

```
origDTThreads <- data.table::setDTthreads(2L)
origNcpus <- options(Ncpus = 2L)

xrange <- yrange <- c(-50, 50)
hab <- terra::rast(terra::ext(c(xrange, yrange)))
hab[] <- 0

# initialize agents
N <- 10

# previous points
x1 <- y1 <- rep(0, N)
# initial points
starts <- cbind(x = stats::runif(N, xrange[1], xrange[2]),
                y = stats::runif(N, yrange[1], yrange[2]))

# create the agent object # the x1 and y1 are needed for "previous location"
agent <- terra::vect(data.frame(x1, y1, starts), geom = c("x", "y"))

ln <- rlnorm(N, 1, 0.02) # log normal step length
sd <- 30 # could be specified globally in params
```

```
if (interactive()) {
  # clearPlot()
  terra::plot(hab, col = "white")
}

for (i in 1:10) {
  agent <- crw(agent = agent, extent = terra::ext(hab), stepLength = ln,
               stddev = sd, lonlat = FALSE, torus = FALSE) # don't wrap
  if (interactive()) terra::plot(agent[, 1], add = TRUE, col = 1:10)
}
terra::crds(agent) # many are "off" the map, i.e., beyond the extent of hab
agent <- SpaDES.tools::wrapTorus(agent, bounds = terra::ext(hab))
terra::plot(agent, add = TRUE, col = 1:10) # now inside the extent of hab

# clean up
data.table::setDTthreads(origDTThreads)
options(Ncpus = origNcpus)
```

Index

.cirSpecialQuick *(*cirSpecialQuick*)*, 14
.pointDistance, 5

adj, 6
adj(), *3*
agentLocation, 9
agentLocation(), *4*
allInRange01, 10

cellFromXY(), *40*, *47*, *57*
cir, 10
cir(), *3*, *16*, *40*, *51*
cirSpecialQuick, 14
crw *(*move*)*, 28
crw(), *4*

directionFromEachPoint(), *3*
distanceFromEachPoint, 15
distanceFromEachPoint(), *3*, *12*
do.call(), *25*
duplicatedInt, 18
dwrpnorm2, 18

fastCrop, 19

gaussMap, 20

heading, 21
heading(), *4*

initiateAgents, 22
initiateAgents(), *4*
inRange, 23

mergeRaster, 24
mergeRaster,list-method *(*mergeRaster*)*, 24
middlePixel, 27
move, 28
move(), *4*

neutralLandscapeMap, 30
neutralLandscapeMap(), *4*
numAgents, 31
numAgents(), *4*, *22*

options(), *5*

parallel::makeCluster(), *16*, *25*
patchSize, 32
pointDistance *(*.pointDistance*)*, 5
probInit, 32
probInit(), *4*, *22*

randomPolygon *(*randomPolygons*)*, 33
randomPolygons, 33
randomPolygons(), *4*, *34*
randomStudyArea, 35
rasterizeReduced, 36
rasterizeReduced(), *4*
rastFromDF, 37
rings, 39
rings(), *3*, *12*, *16*, *51*
runifC, 41

SpaDES.tools *(*SpaDES.tools-package*)*, 3
SpaDES.tools-package, 3
specificNumPerPatch, 42
specificNumPerPatch(), *4*
splitRaster *(*mergeRaster*)*, 24
spokes, 43
spokes(), *3*
spread, 46
spread(), *3*, *5*, *33*, *34*, *39*, *51*, *56*, *62*, *63*, *70*, *72–74*
spread2, 56
spread2(), *5*, *46*, *48*, *51*, *62*, *66*
spread3, 65
spread3(), *48*
spreadCpp, 70
spreadCpp(), *5*, *50*, *61*

terra::adjacent(), *3*, *8*

terra::as.data.frame(), *38*
terra::buffer(), *50*
terra::crop(), *20*
terra::distance(), *15, 16*, *29*, *51*
terra::focal(), *50*
terra::merge(), *25*
terra::mosaic(), *25*
terra::ncell(), *38*
terra::rast(), *37, 38*
terra::SpatRaster, *38*
terra::values(), *38*
testEquivalentMetadata, 74
transitions, 75

wrap *(*wrapTorus*)*, 75
wrapTorus, 75
wrapTorus(), *3*, *29*, *76*